Learning Queuing Networks by Recurrent Neural Networks

Giulio Garbi
giulio.garbi@imtlucca.it
IMT School for Advanced Studies Lucca
Lucca, Italy

Emilio Incerto
emilio.incerto@imtlucca.it
IMT School for Advanced Studies Lucca
Lucca, Italy

Mirco Tribastone
mirco.tribastone@imtlucca.it
IMT School for Advanced Studies Lucca
Lucca, Italy

ABSTRACT

It is well known that building analytical performance models in practice is difficult because it requires a considerable degree of proficiency in the underlying mathematics. In this paper, we propose a machine-learning approach to derive performance models from data. We focus on queuing networks, and crucially exploit a deterministic approximation of their average dynamics in terms of a compact system of ordinary differential equations. We encode these equations into a recurrent neural network whose weights can be directly related to model parameters. This allows for an interpretable structure of the neural network, which can be trained from system measurements to yield a white-box parameterized model that can be used for prediction purposes such as what-if analyses and capacity planning. Using synthetic models as well as a real case study of a load-balancing system, we show the effectiveness of our technique in yielding models with high predictive power.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Modeling and simulation**; • **Software and its engineering** → **Software system models**; **Software performance**.

KEYWORDS

software performance, queuing networks, recurrent neural networks

ACM Reference Format:
Giulio Garbi, Emilio Incerto, and Mirco Tribastone. 2020. Learning Queuing Networks by Recurrent Neural Networks. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3358960.3379134

1 INTRODUCTION

Motivation. Performance metrics such as throughput and response time are important factors that impact on the quality of a software system as perceived by users. They indicate how well the software behaves, thus complementing functional properties that concern what the software does. A traditional way of reasoning about the performance in a software system is by means of profiling. A tool such as *Gprof* executes the program and allows the identification of the program locations that are most performance sensitive [23]. The main limitation is that this information is valid for the specific run with which the program is exercised; different inputs lead to different performance profiles in general. Thus, while profiling can detect the presence of performance anomalies, it lacks generalizing and predictive power (see also [64]).

As with all scientific and engineering disciplines, predictions can be made with models. Software performance models are mathematical abstractions whose analysis provides quantitative insights into real systems under consideration [15]. Typically, these are stochastic models based on Markov chains and other higher-level formalisms such as queueing networks, stochastic process algebra, and stochastic Petri nets (see, e.g., [15] for a detailed account). Although they have proved effective in describing and predicting the performance behavior of complex software systems (e.g., [8, 50]), a pressing limitation is that the current state of the art hinges on considerable craftsmanship to distill the appropriate abstraction level from a concrete software system, and relevant mathematical skills to develop, analyze, and validate the model. Indeed, the amount of knowledge required in both the problem domain and in the modeling techniques necessarily hinders their use in practice [62].

Despite the promises that analytical performance modeling holds, we are confronted with a high adoption barrier. A possible solution might be to derive the model *automatically*. There has been much research into extending higher-level descriptions such as UML diagrams with performance annotations (using for example appropriate profiles such as MARTE [42]) from which both software artifacts and associated performance models are generated (see the surveys [6, 36]). However, since systems are typically subjected to further modifications, the hard problem of keeping the model synchronized with the code arises [20]. This makes such model-driven approaches particularly difficult to use in general, especially in the context of fast-paced software processes characterized by continuous integration and development.

Main contribution. In this paper we propose a novel methodology where analytical performance models are automatically learned from a running system using execution traces. We focus on queueing networks (QNs), a formalism that has enjoyed considerable attention in the software performance engineering community, since it has been shown to be able to capture main performance-related phenomena in software architectures [2], annotated UML diagrams [7], component-based systems [36], web services [18], and adaptive systems [3, 29].

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-6991-6/20/04…$15.00
https://doi.org/10.1145/3358960.3379134

A QN is characterized by a number of parameters that define the following quantities: i) the behavior of each shared resource, such as its service demand and the concurrency level, which describe the amount of time that a client spends at the resource and the number of independent entities that can provide the service (e.g., number of threads in the pool or number of CPU cores), respectively; ii) the behavior of clients in terms of their operational profile, i.e., how they traverse the resources.

Some of these parameters can be assumed to be known. For instance, the number of CPU cores is available from the hardware specification (or from the virtual-machine settings in a virtualized environment); the number of worker threads is a configuration parameter in most servers. Other parameters are more difficult to identify: the service demands, which depend on the execution behavior of the program that requests access to a shared resource; and the *routing matrix*, which defines how clients (probabilistically) move between queuing stations.

In our approach, the input is the set of shared resources and their concurrency level. The objective is to discover the QN model, i.e., the topology of the network and the service demands.

Obviously, the problem of learning a mathematical model from data is not new. In the specific case of identifying parameters of a QN, a substantial amount of research gone into the problem of estimating service demands only ([49], see Section 5.3 for a more detailed account of related work). Instead, we are not aware of approaches that deal with the estimation of both the service demands and the topology. This setting is a rather difficult one from a mathematical viewpoint because, as will be formalized later, routing probabilities and service demands appear as multiplicative factors in the dynamical equations that describe the evolution of a QN [8]. Since learning a QN can be understood as fitting the parameters to match these equations by some form of optimization, using both routing probabilities and service demands as decision variables will induce a *nonlinear problem*, which is very difficult to handle in general. An additional problem to nonlinearity is that of scalability. This is due to the issue that the exact dynamical equations of a QN incur the well-known state explosion problem, because the number of discrete states to keep track of grows combinatorially with the number of clients and queuing stations.

Learning method: recurrent neural networks. To cope with both issues, we propose a learning method based on recurrent neural networks (RNNs) because of their ability to fitting nonlinear systems [41]. In particular, we develop a new architecture of the RNN which encodes the QN dynamics in an *interpretable fashion*, i.e., by associating the weights of the RNN with QN parameters such as concurrency levels, routing probabilities, and service rates. A key instrument is the use of a compact system of *approximate* (but still nonlinear) equations of the QN dynamics instead of the combinatorially large, but exact, original system of equations. Such approximation—called *fluid* or *mean-field*—consists in only one ordinary differential equation (ODE) for each station. It describes the time evolution of the *queue length*, i.e., the number of clients contending for that resource. In practice, the fluid approximation provides an estimate of the average queue length of the underlying stochastic process. The QN approximation procedure is based on a fundamental result by Kurtz [37] and is well-known in the literature, e.g., [9]. In the field of software performance, it has been used for the analysis of variability-intensive software systems [34, 35] and for model-based runtime software adaption using online optimization [29] or satisfiability modulo theory approach [28]. This formulation has also been recently adopted for learning, but for service demands only [27], thus casting the problem into a (considerably) simpler quadratic programming one.

The connection between RNN and ODEs is not new in the literature. In [44], authors have shown that recurrent neural networks can be thought of as a discretization of the continuous dynamical systems while, in [13] a specialized training algorithm for ODEs has been recently proposed. However, despite the proliferation of works along this research direction, still there is no clear understanding of how to employ such artificial intelligence/machine learning techniques for supporting performance engineering tasks such as modeling, estimation, and optimization [38].

> The main technical contribution of this paper is to show that there is a direct association between the structure of the QN fluid approximation and standard activation functions and layers of an RNN. To the best of our knowledge, this is the first approach that formally unifies the expressiveness of analytical performance models with the learning capability of machine learning, contributing to positively answering the question whether "*AI will be at the core of performance engineering*" [38].

The RNN is trained using time series of measured queue lengths at each service station. Its learned weights can be interpreted back as a QN with learned parameters, which can be used for predictive purposes. It is worth remarking that, in principle, one could learn a QN model by relying on a standard, *black-box* RNN architecture by treating all the QN parameters (i.e., initial population, service demand, number of servers and routing probabilities) as input features of the learning algorithm. Unfortunately, this straightforward approach would require a considerable amount of input traces since the learning algorithm could not exploit the structural information about the problem. For instance, it would not be possible to do accurate what-if analyses by varying the value of a parameter if the network had not been trained with some input configurations where few variations of that parameter are considered. Moreover, in such setting, it would even be unclear which weights must be altered and how to reflect the changes into the model.

Instead, here we report on the effectiveness and the generalizing power of our method by considering both synthetic benchmarks on randomly generated QNs, as well as a real web application deployed according to the load balancing architectural style. In both cases, we evaluate the degree of predictive power of the learned model in matching the transient as well as steady-state dynamics of unseen configurations (i.e., by varying the system workload, number of servers, and routing probabilities), reporting prediction errors less than 10% across a validation set of 2000 instances.

Paper organization. We provide some background about QNs in Section 2. The learning methodology is presented in Section 3, which discusses how to encode a time-discretized version of the fluid approximation into an RNN where the weights represent the model parameters to identify. Section 4 presents the numerical evaluation on both the synthetic benchmarks and the real case

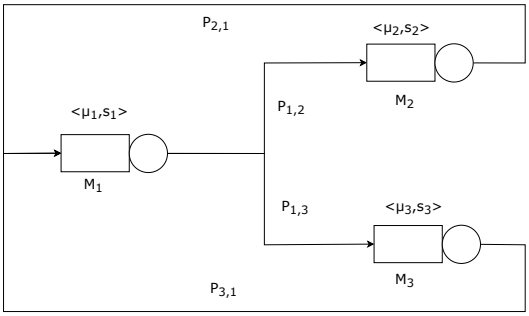


Figure 1: Load balancing example

study, providing implementation details on the RNN and on the used benchmark application. Section 5 discusses further related work. Section 6 concludes.

2 BACKGROUND

To make the paper self-contained, we present some background on QNs with the objective of motivating the fluid approximation as a deterministic estimator of average queue lengths, which will be used for the RNN encoding.

2.1 Queuing Networks

We assume *closed* QNs, where clients keep circulating between queuing stations. A closed QN is formally defined by the following:

- N: the number of clients in the network;
- M: the number of queuing stations;
- $\mathbf{s} = (s_1, \ldots, s_M)$: the vector of concurrency levels, where s_i gives the number of independent servers at station i, with $1 \leq i \leq M$;
- $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_M)$: the vector of service rates, i.e., $1/\mu_i > 0$ is the mean service demand at station i, with $1 \leq i \leq M$;
- $\mathbf{P} = (\mathbf{P}_{i,j})_{1 \leq i,j \leq M}$: the routing probability matrix, where each element $\mathbf{P}_{i,j} \geq 0$ gives the probability that a client goes to station j upon completion at station i;
- $\mathbf{x}(0) = (\mathbf{x}_1(0), \ldots, \mathbf{x}_M(0))$: the *initial condition*, i.e., $\mathbf{x}_i(0)$ is the number of clients at station i at time 0.

In a closed QN, the routing probability matrix is *stochastic* matrix, meaning that the sum across each row sums up to one.

Example 2.1. In the remainder of this section we use the QN in Fig. 1 as a running example. Depicted using the customary graphical representation, it represents a simple load-balancing system with $M = 3$ stations. Requests from reference station ***1*** are routed to two compute server stations ***2*** and ***3*** with probabilities $\mathbf{P}_{1,2}$ and $\mathbf{P}_{1,3}$, respectively. Upon service, a client returns back to station ***1***. An instantiation of this abstract model is discussed in Section 4. □

Markov chain semantics. The stochastic behavior of a QN is represented by a continuous-time Markov chain (CTMC) that tracks the probability of the QN having a given configuration of the queue lengths at each station. Informally, the CTMC is constructed as follows. A discrete CTMC state is a vector of queue lengths $\mathbf{X} = (X_1, \ldots, X_M)$. At each station i, if the number of clients X_i is less than or equal to the number of servers s_i, then these proceed in parallel, each at rate μ_i. Instead, if $X_i > s_i$ the number of clients that are queueing for service is $X_i - s_i$. When one client is serviced at station i, with probability p_{ij} it goes to station j to receive further service. This can be formalized by considering the well-known model of Markov population processes, whereby the CTMC transitions are described by jump vectors and associated transition functions from a generic state $\mathbf{X}$ [9].

We define the jump vectors $h^{(ij)}$ to be the state updates due to clients moving to station j upon service at i, and $q(\mathbf{X}, \mathbf{X} + h^{(ij)})$ the transition rate from state $\mathbf{X}$ to state $\mathbf{X} + h^{(ij)}$, where

$$\mathbf{X} + h^{(ij)} = (X_1, \ldots, X_i - 1, \ldots, X_j + 1, \ldots, X_M).$$

In other words, with the jump vector $h^{(ij)}$ we have that the number of clients at station i is decreased by one, and, correspondingly, the number of clients at station j is increased by one. Then, the CTMC is defined by:

$$q(\mathbf{X}, \mathbf{X} + h^{(ij)}) = \mathbf{P}_{i,j}\mu_i \min(X_i, s_i), \quad i, j = 1, \ldots, M. \tag{1}$$

Example 2.2. In our running example, we have the jump vectors

$$h^{(12)} = (-1, +1, 0) \qquad h^{(13)} = (-1, 0, +1)$$
$$h^{(21)} = (+1, -1, 0) \qquad h^{(31)} = (+1, 0, -1)$$

where the first row describes the updates due to a client being assigned to each compute server and the second row defines the client returning to the load balancer after service. For completeness we give the corresponding transitions:

$$q(\mathbf{X}, \mathbf{X} + h^{(12)}) = \mathbf{P}_{1,2}\mu_1 \min(X_1, s_1)$$
$$q(\mathbf{X}, \mathbf{X} + h^{(13)}) = \mathbf{P}_{1,3}\mu_1 \min(X_1, s_1)$$
$$q(\mathbf{X}, \mathbf{X} + h^{(21)}) = \mathbf{P}_{2,1}\mu_2 \min(X_2, s_2)$$
$$q(\mathbf{X}, \mathbf{X} + h^{(31)}) = \mathbf{P}_{3,1}\mu_3 \min(X_3, s_3)$$

□

It is well known that a CTMC is completely characterized by the transitions (1) together with the initial condition $\mathbf{x}(0)$. This formulation in terms of jump vectors allows for the efficient stochastic simulation of CTMCs [22]; indeed, we will use this technique to generate sample paths for the evaluation of our learning method on synthetic benchmarks in Section 4. For our purposes, the main limitation of this CTMC representation is that the exact equations to analyze the probability distribution grow combinatorially with the number of clients and stations, as one needs to keep track of each possible discrete configuration of the queue lengths.

2.2 Fluid Approximation

The fluid approximation of a QN consists is an ODE system whose size is equal to the number of stations M, independently from the number of clients in the system. Informally, the ODE system can be built by considering the average impact that each transition has on the queue length at each station k. This is obtained by multiplying the k-th coordinate of each jump vector, $h_k^{(ij)}$, by the function associated with the corresponding transition rate $q(\mathbf{X}, \mathbf{X}+h^{(ij)})$. Denoting by $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_M)$ the variables of the fluid approximation,

the ODE system is given by:

$$\frac{d\mathbf{x}_k(t)}{dt} = \sum_{h^{(ij)}} h_k^{(ij)} q(\mathbf{x}(t), \mathbf{x}(t) + h^{(ij)}),\ k = 1, \ldots, M. \quad (2)$$

The solution for each coordinate, $\mathbf{x}_k(t)$, can be interpreted as an approximation of the average queue length at time t as given by the CTMC semantics [9]. The theorems in [37] provide a result of asymptotic exactness of the fluid approximation, in the sense that the ODE solution and the expectation of the stochastic process become indistinguishable when the number of clients and servers is large enough.

Using (1), the equations can be written as follows:

$$\frac{d\mathbf{x}_k(t)}{dt} = \sum_{i \neq k} \mathbf{P}_{i,k} \boldsymbol{\mu}_i \min(\mathbf{x}_i(t), \mathbf{s}_i) + (\mathbf{P}_{k,k} - 1)\boldsymbol{\mu}_k \min(\mathbf{x}_k(t), \mathbf{s}_k) \quad (3)$$

where we have singled out the rates due to self loops $\mathbf{P}_{k,k}$.

Example 2.3. The fluid approximation for the load balancer is:

$$\begin{aligned}
\frac{d\mathbf{x}_1(t)}{dt} &= -\boldsymbol{\mu}_1 \min(\mathbf{x}_1(t), \mathbf{s}_1) + \boldsymbol{\mu}_2 \min(\mathbf{x}_2(t), \mathbf{s}_2) + \\
&\quad + \boldsymbol{\mu}_3 \min(\mathbf{x}_3(t), \mathbf{s}_3) \\
\frac{d\mathbf{x}_2(t)}{dt} &= -\boldsymbol{\mu}_2 \min(\mathbf{x}_2(t), \mathbf{s}_2) + \mathbf{P}_{1,2}\boldsymbol{\mu}_1 \min(\mathbf{x}_1(t), \mathbf{s}_1) \\
\frac{d\mathbf{x}_3(t)}{dt} &= -\boldsymbol{\mu}_3 \min(\mathbf{x}_3(t), \mathbf{s}_3) + \mathbf{P}_{1,3}\boldsymbol{\mu}_1 \min(\mathbf{x}_1(t), \mathbf{s}_1)
\end{aligned}$$

□

Based on the solution to Eq. 3, which directly provides queue-length estimates, one can derive other important performance metrics such as throughput, utilization, and response time. See, for instance [56, 57] for a study of these results in a process algebra [25], and [54, 55, 58] for applications to layered queueing networks [19].

In the remainder of this paper, we shall focus on QNs that do not have *self loops* (i.e., a client served at a queue cannot re-enter the same queue immediately), i.e., $\mathbf{P}_{i,i} = 0$ for $1 \leq i \leq M$. This is because we can show that, in the fluid approximation, for each k, $\mathbf{P}_{k,k}$ can be chosen freely as long as we adjust each $\mathbf{P}_{k,i}$ with $i \neq k$ and $\boldsymbol{\mu}_k$. More formally, we can prove the following theorem.

Theorem 2.1. *For each $\pi \in [0,1)^M$, stochastic matrix $\mathbf{P}$ and $\boldsymbol{\mu} \geq 0$ where* (3) *holds, there exist $\hat{\mathbf{P}}$ and $\hat{\boldsymbol{\mu}}$ such as for each k:*

(a) $\frac{d\mathbf{x}_k(t)}{dt} = \sum_{i \neq k} \hat{\mathbf{P}}_{i,k}\hat{\boldsymbol{\mu}}_i \min(\mathbf{x}_i(t), \mathbf{s}_i) + (\hat{\mathbf{P}}_{k,k} - 1)\hat{\boldsymbol{\mu}}_k \min(\mathbf{x}_k(t), \mathbf{s}_k)$;
(b) $\hat{\mathbf{P}}_{k,k} = \pi_k$;
(c) $\sum_i \hat{\mathbf{P}}_{k,i} = 1$;
(d) $\forall i\ \hat{\mathbf{P}}_{k,i} \geq 0$;
(e) $\hat{\boldsymbol{\mu}}_k \geq 0$.

Proof. Available in Appendix A. □

Thus, using the fluid approximation, for each network with self loops there is another one without them which cannot be distinguished. To identify a specific network among them, we need to know the self-loop values.

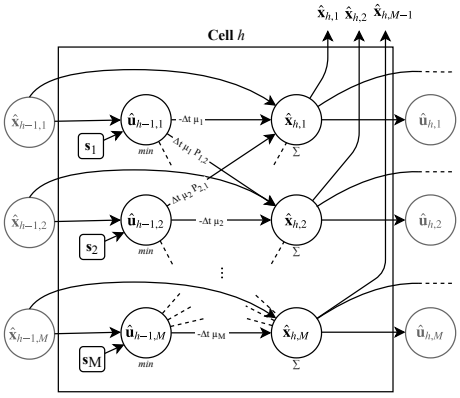


Figure 2: RNN encoding

3 LEARNING METHODOLOGY

As apparent in both Equations (1) and (3), a QN features routing probabilities and service demands as multiplicative factors in the defining dynamical equations. If we wish to learn a QN by assuming that both quantities are unknown, we are faced with a nonlinear (i.e., polynomial) optimization problem. Here we propose an RNN in order to estimate these parameters. We develop an RNN architecture which encodes the QN dynamics in an *interpretable fashion*, i.e., by associating the weights of the RNN with QN parameters such as concurrency levels, routing probabilities, and service rates.

3.1 ODE Discretization

We first obtain a time-discrete representation of the fluid approximation such that each time step is associated with a layer of the RNN. In matrix notation, for an arbitrary QN the fluid approximation is given by:

$$\frac{d\mathbf{x}(t)}{dt} = -\boldsymbol{\mu} \min(\mathbf{x}(t), \mathbf{s}) + \mathbf{P}^T \boldsymbol{\mu} \min(\mathbf{x}(t), \mathbf{s})$$

where $\mathbf{x}(t)$ is the M-dimensional vector of queue lengths at time t. We consider a finite-step approximation of the above ODE for a small Δt, obtaining:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \big(-\boldsymbol{\mu} \min(\mathbf{x}(t), \mathbf{s}) + \boldsymbol{\mu}\mathbf{P} \min(\mathbf{x}(t), \mathbf{s})\big)$$

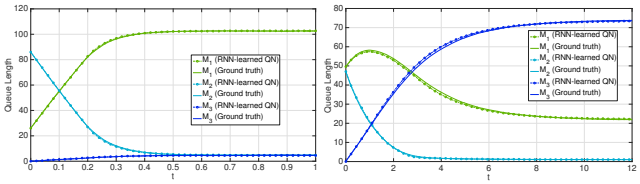
Finally, this can be rewritten as

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \cdot \mathbf{u}(t) \cdot (\boldsymbol{\mu} \odot (\mathbf{P} - \mathbf{I})) \quad (4)$$

where $\mathbf{u}(t) = \min(\mathbf{x}(t), \mathbf{s})$, $\mathbf{I}$ is the identity matrix of appropriate dimension, and $\odot$ is the operator where if $\mathbf{C} = \mathbf{a} \odot \mathbf{B}$, then $\mathbf{C}_{i,j} = \mathbf{a}_i \cdot \mathbf{B}_{i,j}$.

3.2 RNN Encoding

The discretization (4) of the fluid approximation of the QN admits a direct encoding as an RNN. It consists of an M-dimensional input layer $\hat{\mathbf{x}}_0$ that corresponds to the initial condition of the QN. The RNN has $H - 1$ cells, with the h-th cell computing the estimate of the queue length at time $h\Delta t$, denoted by $\hat{\mathbf{x}}_h$ (see Fig. 2). That is, the h-th cell computes the quantity $\hat{\mathbf{x}}_h = \hat{\mathbf{x}}_{h-1} + \Delta t \cdot \hat{\mathbf{u}}_{h-1} \cdot (\boldsymbol{\mu} \odot (\mathbf{P} - \mathbf{I}))$, where, according to (4), $\hat{\mathbf{u}}_{h-1}$ estimates $\mathbf{u}((h-1)\Delta t)$ as $\hat{\mathbf{u}}_{h-1} = \min(\mathbf{s}, \hat{\mathbf{x}}_{h-1})$.



(a) (b)

Figure 3: Numerical evaluation of the running example (see Figure 1). Comparison between simulations of the queue lengths using the RNN-learned QN (marked lines) and the ground-truth one (straight lines) in two different cases: a) a trace used for training ($err = 0.69\%$) with initial population $x(0) = (26, 86, 0)$ and concurrency levels (s_1=1000, s_2=30, s_3=25) b) what-if analysis under unseen initial population vector and unseen concurrency levels (s_1=1000, s_2=6, s_3=1) and initial population $x(0) = (49, 47, 0)$, causing a significant change in the dynamics ($err = 1.49\%$).

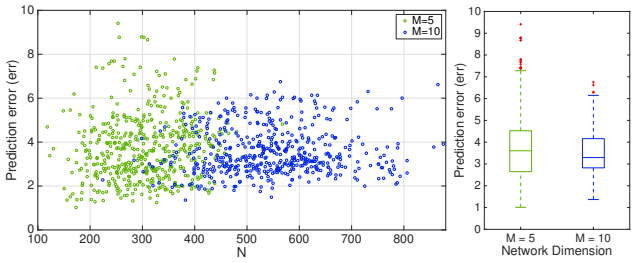
With this set up, we will have to learn the matrix $\mathbf{P}$ (made of $M(M-1)$ weights, since the diagonal is empty) and the vector $\boldsymbol{\mu}$ (made of M weights).

The main goal of this methodology is to learn the actual parameters of the network. Therefore, we enforce some feasibility constraints, namely we require that $\mathbf{P}$ rows sum up to 1 (such that $\mathbf{P}$ is a stochastic matrix), absence of self loops and $\boldsymbol{\mu} \geq 0$ (such that the speed of the stations is non-negative). The non-negativity of the weights is enforced in the framework by clamping the candidate values within the range $[0, \infty)$; stochasticity of $\mathbf{P}$ is guaranteed by dividing each weight by the sum of the weights in the corresponding row; the absence of self loops is achieved by setting $\forall i, \mathbf{P}_{i,i} = 0$ as a constant. This approach puts our work in the *explainable machine learning* research area [45], and it allows us to link each learned parameter with its role in the system. This link allows us to predict the behavior of the system under new conditions (*what-if* analysis). In contrast, a traditional approach to neural networks would not impose a model and constraints on the parameters, hence giving a read-only model which cannot be clearly interpreted. Indeed, without a direct association between parameters and physical quantities, we cannot study the system under new conditions unless learning a new model.

Example 3.1. The RNN encoding for the h-th cell (i.e., the queue length transient evolution at time $h\Delta t$) of our running example is:

$$\hat{\mathbf{u}}_{h-1,1} = \max\left(\mathbf{s}_1, \hat{\mathbf{x}}_{h-1,1}\right)$$
$$\hat{\mathbf{u}}_{h-1,2} = \max\left(\mathbf{s}_2, \hat{\mathbf{x}}_{h-1,2}\right)$$
$$\hat{\mathbf{u}}_{h-1,3} = \max\left(\mathbf{s}_3, \hat{\mathbf{x}}_{h-1,3}\right)$$
$$\hat{\mathbf{x}}_{h,1} = \hat{\mathbf{x}}_{h-1,1} + \Delta t\left(-\boldsymbol{\mu}_1\hat{\mathbf{u}}_{h-1,1} + \boldsymbol{\mu}_2\mathbf{P}_{2,1}\hat{\mathbf{u}}_{h-1,2} + \boldsymbol{\mu}_3\mathbf{P}_{3,1}\hat{\mathbf{u}}_{h-1,3}\right)$$
$$\hat{\mathbf{x}}_{h,2} = \hat{\mathbf{x}}_{h-1,2} + \Delta t\left(\boldsymbol{\mu}_1\mathbf{P}_{1,2}\hat{\mathbf{u}}_{h-1,1} - \boldsymbol{\mu}_2\hat{\mathbf{u}}_{h-1,2} + \boldsymbol{\mu}_3\mathbf{P}_{3,2}\hat{\mathbf{u}}_{h-1,3}\right)$$
$$\hat{\mathbf{x}}_{h,3} = \hat{\mathbf{x}}_{h-1,3} + \Delta t\left(\boldsymbol{\mu}_1\mathbf{P}_{1,3}\hat{\mathbf{u}}_{h-1,1} + \boldsymbol{\mu}_2\mathbf{P}_{2,3}\hat{\mathbf{u}}_{h-1,2} - \boldsymbol{\mu}_3\hat{\mathbf{u}}_{h-1,3}\right)$$

□



(a) (b)

Figure 4: a) Prediction error of the what-if instances where each randomly generated QN is tested with 100 unseen initial population vectors, distinguished in colors with respect to the network size M. The x-axis N is the total number of clients in the network which is scatter-plotted against the prediction error defined in Eq. (5). b) Statistics on the prediction error. In each box-plot, the line inside the box represents the median error, the upper and lower side of the box represent the 25th and 75th percentiles, while the upper and lower limit of the dashed line represent the extreme points not to be considered outliers, and in red we depict the outliers (12 with M=5, 4 with M=10).

3.3 Input data

The RNN is trained over a set of traces. Each trace is made of H vectors, indicated as $\tilde{\mathbf{x}}_0, \tilde{\mathbf{x}}_1, ..., \tilde{\mathbf{x}}_{H-1} \in \mathbb{R}^M_{\geq 0}$. The i-th component $\tilde{\mathbf{x}}_{h,i}$ of each vector $\tilde{\mathbf{x}}_h$ represents a sample of the queue length of each station i at time $h \cdot \Delta t$. Since, as discussed, the fluid approximation can be interpreted as an estimator of the average queue lengths, each trace used in the learning process consists of measurements averaged over a number of independent executions started with the same initial condition; different traces rely on different initial conditions to exercise distinct behaviors of the system.

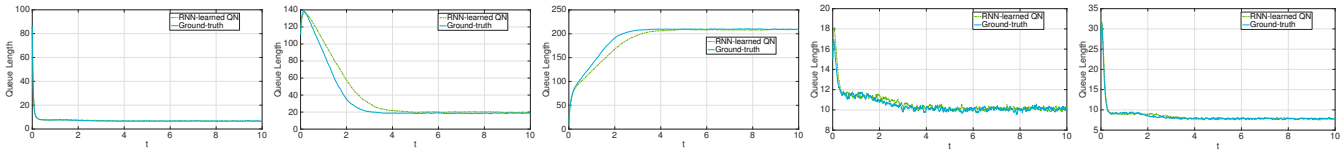
3.4 Learning function

The learning error function, denoted by *err*, aims to minimize the difference between the queue lengths estimated by the RNN, $\hat{\mathbf{x}}_h$, and the measurements $\tilde{\mathbf{x}}_h$. It is defined as follows:

$$err = \frac{\max_{h=1}^{H-1} \|\tilde{\mathbf{x}}_h - \hat{\mathbf{x}}_h\|}{2N} \cdot 100 \tag{5}$$

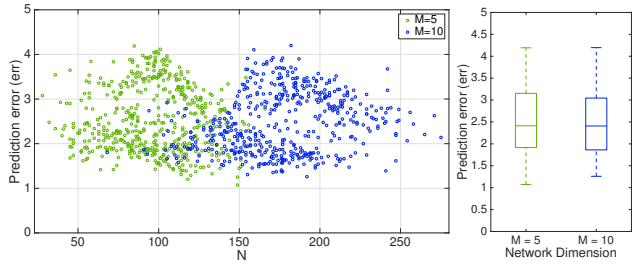
where $\|\cdot\|$ indicates the L1 norm. Essentially, it is a maximum relative error. Indeed, since we are studying closed QNs with fixed N circulating clients, the quantity $\|\tilde{\mathbf{x}}_h - \hat{\mathbf{x}}_h\|/(2N)$ intuitively measures the proportion of clients (relative to their total number N) that are "misplaced" (i.e., which are allocated in a different station) at each time step. Since a misplaced client is counted twice (once when missing in a queue and once when is extra in another queue), we divide the norm by 2. Then, the overall error *err* computes the maximum of such misplacements across all times.

Example 3.2. Let us consider our running example by fixing ground-truth parameters as follows. During the learning phase, we studied the system with $\mathbf{s} = (1000, 30, 25)$ and predicted the



(a) Station 1 (b) Station 2 (c) Station 3 (d) Station 4 (e) Station 5

Figure 5: Comparison between the ground-truth queue lengths and those predicted by the RNN-learned QN on the test case that induced the maximum prediction error among the what-if over population (error: 9.41%). The error was attained on a randomly generated QN with $M = 5$ stations, using the unseen initial population vector (86,111,13,15,28). The straight line represents the ground-truth dynamics of the QN model; the dashed line represents the evolution of the RNN-learned QN.



(a) (b)

Figure 6: a) Prediction error of the what-if instances by changing the concurrency level of the most utilized station in each of the randomly generated QNs. b) Statistics on the prediction error.

behavior with $\mathbf{s} = (1000, 6, 1)$, while we kept $\mathbf{P}$ and $\boldsymbol{\mu}$ unchanged at

$$\mathbf{P} = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad \boldsymbol{\mu} = (1, 11, 11)$$

Using the experimental set up that will be discussed in the next section, we generated the training dataset by collecting 50 traces, one for a different randomly generated initial population vector. Each trace was the average of 500 independent simulations recording the transient evolutions of the queue lengths. Figure 3 reports the comparison between queue lengths of the RNN-learned QN and the ground-truth one, showing very good accuracy on an instance of the training set (Figure 3a) as well as high predictive power of the model under unseen initial populations and concurrency levels which cause bottleneck shift and considerable longer transient dynamics (Figure 3b).

4 NUMERICAL EVALUATION

In this section we evaluate the effectiveness of the proposed approach by considering both synthetic benchmarks and a real case study. For all our tests, the RNNs were implemented using the Keras framework [14] with the TensorFlow backend [1]. Learning was performed using a machine running the 4.15.0-55-generic Linux kernel on a Intel(R) Xeon(R) CPU E7-4830 v4 machine at 2.00GHz with 500 GB of RAM.

4.1 Synthetic case studies

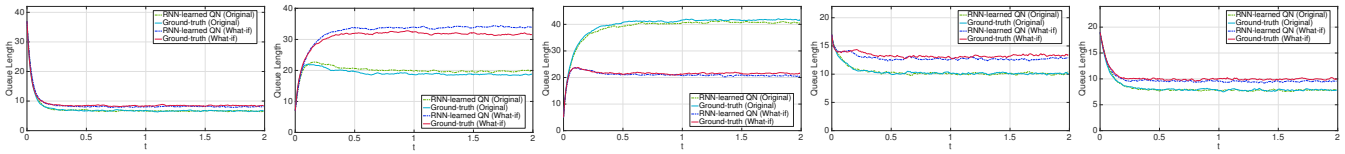
Set-up. For our synthetic tests we considered randomly generated networks of size $M = 5$ and $M = 10$. For each case, we generated 5 QNs by uniformly sampling at random the entries of the routing probability matrices, the service rates in the interval $[4.0, 30.0]$, and the concurrency levels in the interval $\{15, 16, \ldots, 30\}$. For the training of each QN, we generated 100 traces, each being the average over 500 independent stochastic simulations (generated using Gillespie's algorithm [22]). Each trace exercised the model with a distinct initial population vector such that the number of clients at each station was drawn uniformly at random from $\{0, \ldots, 40\}$; as a result, the total number of clients in the network varies across traces. For each network, learning was performed by equally splitting the 100 traces for training and validation, iterating Adam [33] with learning rate equal to 0.05, until the error computed on the validation set did not improve by at least 0.01% in the last 50 iterations. On average, the learning took 74 minutes and 86 minutes for the cases $M = 5$ and $M = 10$, respectively.

Discretization methodology. Two important parameters are the length of the trace, i.e., the time horizon T of the stochastic simulations, and the choice of the discretization interval Δt; these are related with the number of cells in the RNN H by $T = (H - 1)\Delta t$. Longer time horizons lead to larger simulation (hence, training) runtimes. Too short traces might not expose the full dynamics of the system. Further, following basic facts about ODE discretization [4], the interval Δt should be chosen small enough such that no important dynamics is lost across two successive time steps; thus, longer time horizons might need more time steps, hence more cells in the RNN. It is worth remarking that these considerations are model-specific. That is, the choice of such hyper-parameters must be carefully done depending on the specific QN under study.

For the synthetic case studies, we set $T = 10$ and $\Delta t = 0.01$, hence $H = 1000$.

Predictive power. We evaluate the predictive power of the learned QNs by performing two distinct "what-if" analyses under unseen configurations, by changing populations of clients and the concurrency levels of the stations, respectively.

What-if analysis over client population. We tested each of the randomly generated QNs with 100 new initial population vectors that were not used in the learning phase. We compared the averages



(a) Station 1 (b) Station 2 (c) Station 3 (d) Station 4 (e) Station 5

Figure 7: Comparison between the ground-truth queue lengths and those predicted by the RNN-learned QN on the test case that induced the maximum prediction error (4.19%), before and after the what-if change of server concurrency. The error was attained on a randomly generated QN with M = 5 stations. The cyan line denotes the averages under the original conditions (before the what-if change) with the ground-truth QN; the green line gives the predictions of the RNN-learned QN with the original values; the red line shows ground-truth simulations with the unseen number of servers for the bottleneck station (increased from 17 to 37); the blue line shows the averages after the what-if change for the RNN-learned QN.

(over 500 stochastic simulations) of the the ground-truth queue-length dynamics with those produced by the RNN-learned QN with those unseen initial conditions.

Figure 4a shows a scatter plot of the prediction error with respect to the total number of clients circulating in the system, reporting errors less than 10% in all cases. The box-plots in Figure 4b show that there is no statistically significant difference between the errors for the diffrent sized models. Figure 5 compares the predicted and ground-truth queue lengths for the instance with the maximum prediction error, showing a very good generalizing power for the queue-length dynamics at all stations.

What-if over concurrency levels. To validate the predictive power under varying concurrency levels, for each generated QN we found the station with the highest ratio between the steady state queue length and its number of servers (*bottleneck*), and added servers in steps of 20 to this station until it was not the bottleneck anymore. Then we compared the dynamics of the ground-truth model (i.e., simulated with the original **P** and $\boldsymbol{\mu}$ but with the new server concurrency levels) against those obtained by simulating the learned model with the new server concurrency levels. We considered the notion of prediction error as shown in Equation (5).

Figure 6a shows the results of this what-if study, reporting a prediction error less than 5% across all instances. Also in this case, there is no statistically significant difference in the error statistics depending on the network size M (see Figure 6b). Figure 7 plots the comparison of the queue-length dynamics of the what-if instance (i.e., with an unseen server concurrency level) that reported the maximum prediction error (i.e., 4.5%) against the original ones (i.e., prediction error of 3.1%). We can appreciate that the unseen concurrency levels do change the QN behavior dramatically, effectively switching the bottleneck from station 3 to station 2.

This result does support the combination of machine learning and white-box performance models by showing that, once learned, the QN can be used for evaluating the behavior of the model under execution scenarios for which the QN has not been trained.

4.2 Real case study

Set-up. The benchmark used in this evaluation is based on an in-house developed web application that serves user requests with an input dependent load. We deployed the target application as a NodeJs [53] load-balancing system with three replicas. Figure 8 (left) depicts the system architecture. Component **W** represents the reference station, where clients enter the system by issuing requests to the load balancer **LB**, which redistributes them across the web servers uniformly. In the real system, such uniform assignment is achieved by fixing equal *weights* to the target nodes. Components **C1**, **C2**, and **C3** represent the three web-server instances devoted to the actual processing of user requests (e.g., producing an HTML page). Each node in the Figure 8 is annotated with its concurrency level (i.e., the number of available processes), which we considered fixed parameters.

Specifically, we implemented **W** as a multi-threaded Python program. Each thread runs an independent concurrent user (i.e., one of the N processes) that iteratively accesses the system, sleeping for an exponentially distributed delay between subsequent requests; **LB** is a single-threaded NodeJs web server which act as a randomized load balancer. Finally, **C1**, **C2** and **C3** are multi-threaded NodeJs Clusters[1] whose load is generated by sleeping for an exponential distributed delay (i.e., the average value is given as input parameter of each cluster). We remark that although we were able to roughly fix the distribution of the service demands their exact shape is still unknown since it is influenced by subtler factors that are hidden to developers (e.g., internal behavior of the web server, communication aspects). Moreover, in order to evaluate our learning methodology in an interesting scenario, we deployed the three replicas of the system with different parallelism levels and different service rates.

Similarly to [61], we collected the queue length traces used as input of the learning process (see Section 3) by parsing the access logs generated by each component of the system. However, other monitoring solutions could be used, based for instance on recording the TCP backlog [29]. With this set-up, we were able to sample data with a measurement step $\Delta t = 0.01$ s, which turned out to be sufficient for observing the transient dynamics of each component without altering the application behavior. The replication package for this evaluation is publicly available at https://zenodo.org/record/3679251.

Model Learning: We built the training dataset as a collection of queue length traces produced by the target application under 50 different initial population vectors where each station had a number of clients drawn uniformly at random between 0 and 30. For each

[1] https://nodejs.org/api/cluster.html

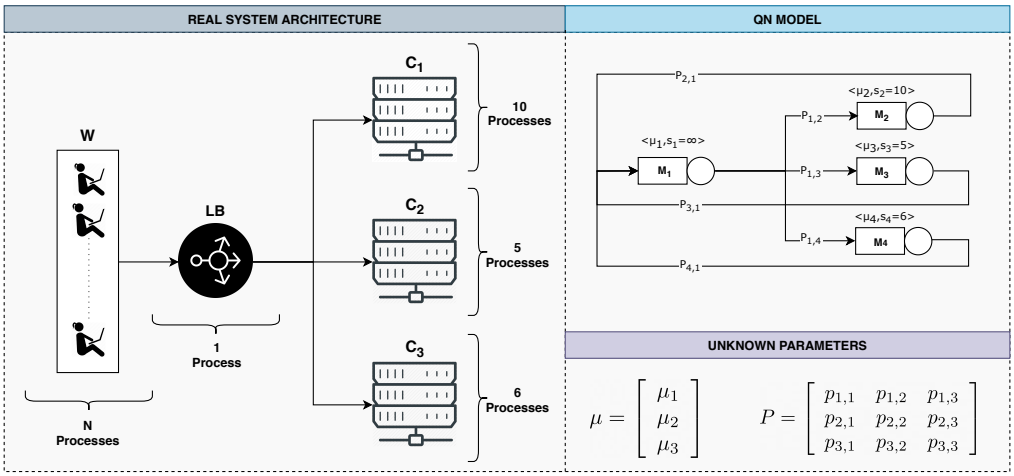
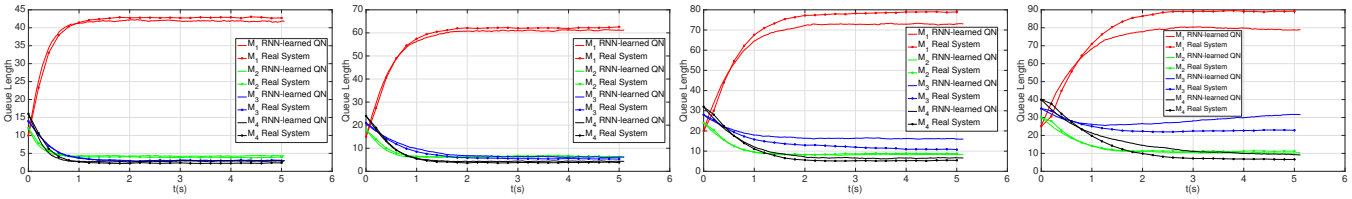


Figure 8: Case study architecture.



(a) $k = 2$, **err=6.46%** **(b)** $k = 3$, **err=5.03%** **(c)** $k = 4$, **err=6.45%** **(d)** $k = 5$, **err=9.05%**

Figure 9: Comparison between the real system dynamics (i.e., marked lines) and the RNN-learned QN (i.e., straight lines) in what-if cases over increasing circulating populations N, given by $N = 26k$.

such initial population vector the trace consisted of the average queue-length dynamics over 500 independent executions.

The target model of the learning process is reported in the right side of Figure 8. In particular, components **C1**, **C2**, **C3** are modeled by queuing stations M_2, M_3, M_4, while both the workload generator **W** and the load balancer **LB** are abstracted by the same station M_1, since the delay introduced by **LB** is negligible with respect to the other components of the network. All the parameters of the resulting QN were considered parameters to be learned by the RNN.

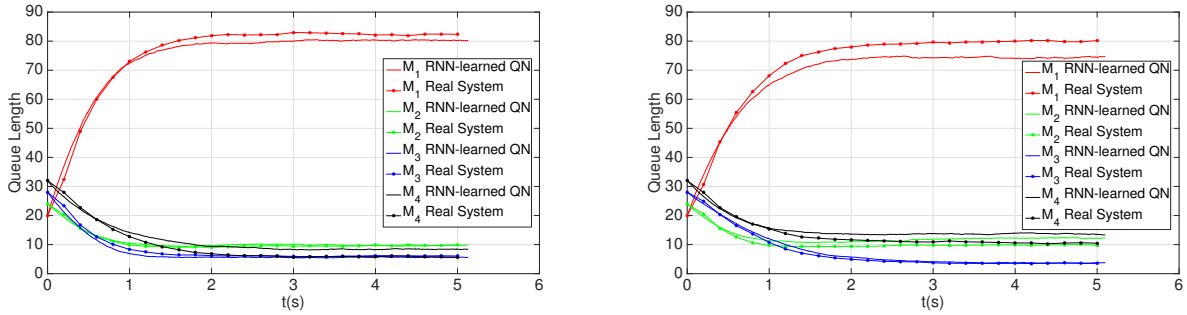
Similarly to the synthetic case, the collected traces were split into two halves for training and validation, respectively. We used Adam [33] as the learning algorithm with learning rate equal to 0.01 and iterated until the error computed on the validation set did not improve at least 0.01% in the last 50 iterations. With this, the system parameters were learned in 27 minutes on average, with a validation error of 3.89%.

What if analysis: In the following we evaluate the predictive power of the RNN learned QN under an unseen number of clients, concurrency levels and routing probabilities. Differently from the synthetic case, here we emulate a concrete usage scenario in which an initially hidden performance bottleneck is discovered and solved only relying on the insights given by the learned model. For doing so, we exercised both the QN model and the real system under an increasing number of clients (here each simulation averaged over 300 simulation runs instead of 500 since for evaluating the what-if analysis less runs are needed) by a factor $k = 2, \ldots, 5$ with respect to an initial population which had 26 circulating clients. Figure 9 reports the numerical results of this evaluation, showing a trend that induces a saturation condition in station M_3. Overall, the prediction error of the RNN is less than 10% across all instances.

In Figure 10 we report two different strategies that can be used in order to remove the bottleneck: we reevaluated both the learned model and the real system starting from the case $k = 4$ (see Figure 9c), varying either the number of servers or the load-balancing weights/routing probabilities. Figure 10a shows the dynamics of the system when the number of servers of M_3 is increased from 5 to 8, Figure 10b reports the what-if scenario in which we change the load distribution strategy from a uniform probability distribution to one where stations M_2, M_3, and M_4 are targeted with probability 0.35, 0.20 and 0.45, respectively. Consistently with the intuition, both what-if instances show a lighter pressure (i.e., smaller queue length) at M_3. Furthermore, both situations are well predicted by the RNN, yielding an accuracy error of ca. 6% with respect to the real system dynamics.

5 RELATED WORK

In this section we relate against techniques related to the following lines of research: performance prediction from programs, generation of performance models from programs, and estimation of parameters in QNs.



(a) err=5.98% **(b) err=6.10%**

Figure 10: a) What-if scenario changing the concurrency level of M_3 from 5 to 8. b) What-if scenario changing the load balancing strategy from a uniform probability distribution to the case where stations M_2, M_3, and M_4 are reached with probabilities 0.35, 0.20 and 0.45, respectively. Both scenarios have been evaluated on the real case study and the RNN-learned QN with an initial population vector with $k = 4$ from Figure 9c.

5.1 Program-driven Performance Prediction

A line of work focuses on the derivation of performance predictions from code analysis. *PerfPlotter* uses program analysis (specifically, probabilistic symbolic execution [21]) to generate a *performance distribution*, i.e., the probability distribution function of a performance metric such as response time [12]. Thus, the result of the overall analysis is a quantitative model but it is not predictive. Furthermore, the approach applies to single-threaded applications, hence important performance-influencing sources such as threads contention cannot be captured.

Other related approaches consist in predicting performance models using *black-box* methods. They are particularly relevant for variability-intensive systems, where they relate configuration settings in a software system with their performance impact [47, 48]. Machine-learning techniques have been used also in this case to build the predictive model [24, 30, 47, 59]. For instance, in [48] the system model is assumed to be a linear combination of binary variables (e.g., tree structured models), each of them denoting the presence or the absence of a feature. Then the performance model is computed by means of linear regression over pairs of configurations and measured performance indices. The influence of possible feature interactions is embedded in the model by introducing fresh variables so as to preserve the linear structure of the model. As discussed in [59], these black-box approaches can be seen as complementary to ours, which can provide a reliable mathematical abstraction by which performance can be explicitly associated to software components, thus increasing the explanatory power of the prediction.

5.2 Program-driven Generation of Performance Models

While model-driven approaches to software performance have been researched quite intensively [15], program-driven generation of performance models has been less explored, and has been concerned with specific kinds of applications. Indeed, the early approach by Hrischuk et al. is concerned with the generation of software performance models (specifically, layered queuing networks [19]) from a class of distributed applications whose components communicate solely by remote procedure calls [26]. Brosig et al. derive a component-based performance model from applications running on the Java EE platform [10, 11]. Tarvo and Reiss develop a technique for the extraction of discrete-event simulation models from a class of multi-threaded programs covering task-oriented applications, whereby the business logic consists in assigning a given workload (i.e., a task) to a number of worker threads from a pool [52]. Their use of a simulation model as opposed to an analytical model is justified by the difficulty in building the latter, especially to model such diverse performance-related phenomena as queuing effects, inter-thread synchronization, and hardware contention. This is indeed the limitation that we aim to overcome with our approach, by building the analytical model automatically from measurements.

5.3 Estimation of service demands in queuing networks

Most of the literature concerning the estimation of QN parameters focuses on service demands. In particular, it considers the situation when the system is in the steady-state, i.e., when a sufficiently large amount of time has passed such that its behavior does not depend on the initial conditions [49]. Mathematically, the assumption of a steady-state regime enables the leveraging of a wealth of analytical results for QNs [8]. Based on these are several estimation methods using techniques such as linear regression [43], quadratic programming [27], non-linear optimization [5, 40], clustering regression [16], independent component analysis [46], pattern matching [17], Gibbs sampling [51, 60], and maximum likelihood [61].

The main advancement of our approach with respect to the state of the art is the ability to learn the whole model, i.e., both the service demands and the QN topology (via the routing probabilities). In addition, since it uses an ODE representation, it does not make assumptions about the stationarity of the system; indeed, we do train our RNN using traces that include the transient dynamics. Actually, our approach uses the same QN model as the service-demand estimation method recently proposed in [27], which is also based on fluid approximation.

Another difference with practical implications regards the type of data using for the estimation. Approaches such as [16, 17, 31, 39, 46] require measurements of quantities that may be difficult to obtain. For example, utilization metrics may not be available to the user when there is no complete information about the underlying hardware stack, for instance in a virtualized system running on a Platform-as-a-Service environment. Instead, measuring queue-length samples only has been regarded as more advantageous [27, 61], since this information can be often obtained from application logs or by means of operating system calls.

6 CONCLUSIONS

We presented a novel methodology for learning queuing network (QN) models of software systems. The main novelty lies in the encoding of the QN as an explainable recurrent neural network where inputs and weights are associated to standard queuing network inputs and parameters. We reported promising results on synthetic examples and on a real case study, where the maximum discrepancy between the dynamics predicted by the learned models and those computed through the ground truth is less than the 10% when the system is evaluated under unseen configurations that are not included in the training set. We plan to extend our technique for capturing more complex models and systems, such as mixed multi-class and layered QNs, and to explore other learning methodologies such as neural ODEs [13] and residual networks [63]. Moreover, in order to improve the accuracy of the learned models and to reduce the simulation time, we plan to investigate active learning techniques that enable an informed sampling of the initial conditions [32].

ACKNOWLEDGMENTS

This work has been partially supported by the PRIN project "SEDUCE" no. 2017TWRCNB.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, and Paul Barham et Al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.
[2] Aldeida Aleti, Barbora Buhnova, Lars Grunske, Anne Koziolek, and Indika Meedeniya. 2013. Software Architecture Optimization Methods: A Systematic Literature Review. *IEEE Trans. Software Eng.* 39, 5 (2013), 658–683. https://doi.org/10.1109/TSE.2012.64
[3] Davide Arcelli, Vittorio Cortellessa, Antonio Filieri, and Alberto Leva. 2015. Control Theory for Model-based Performance-driven Software Adaptation. In *QoSA*. 11–20. https://doi.org/10.1145/2737182.2737187
[4] Uri M. Ascher and Linda R. Petzold. 1988. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM.
[5] Mahmoud Awad and Daniel A. Menasce. 2017. Deriving Parameters for Open and Closed QN Models of Operational Systems Through Black Box Optimization. In *ICPE*.
[6] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. 2004. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Software Eng.* 30, 5 (2004), 295–310.
[7] Simonetta Balsamo and Moreno Marzolla. 2005. Performance evaluation of UML software architectures with multiclass Queueing Network models. In *WOSP*.
[8] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Trivedi. 2005. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley.
[9] Luca Bortolussi, Jane Hillston, Diego Latella, and Mieke Massink. 2013. Continuous approximation of collective system behaviour: A tutorial. *Performance Evaluation* 70, 5 (2013), 317–349.
[10] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. 2011. Automated extraction of architecture-level performance models of distributed component-based systems. In *ASE*.
[11] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. 2009. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *VALUETOOLS*.
[12] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating performance distributions via probabilistic symbolic execution. In *ICSE*.
[13] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. 2018. Neural ordinary differential equations. In *Advances in neural information processing systems*. 6571–6583.
[14] François Chollet et al. 2015. Keras. https://keras.io.
[15] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. 2011. *Model-Based Software Performance Analysis*. Springer.
[16] Paolo Cremonesi, Kanika Dhyani, and Andrea Sansottera. 2010. Service time estimation with a refinement enhanced hybrid clustering algorithm. In *International Conference on Analytical and Stochastic Modeling Techniques and Applications*. Springer, 291–305.
[17] Paolo Cremonesi and Andrea Sansottera. 2014. Indirect estimation of service demands in the presence of structural changes. *Performance Evaluation* 73 (2014), 18–40.
[18] A. Di Marco and P. Inverardi. 2004. Compositional generation of software architecture performance QN models. In *WICSA 2004*. 37–46. https://doi.org/10.1109/WICSA.2004.1310688
[19] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. 2009. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Trans. Software Eng.* 35, 2 (2009), 148–161.
[20] Joshua Garcia, Ivo Krka, Chris Mattmann, and Nenad Medvidovic. 2013. Obtaining ground-truth software architectures. In *ICSE*. 901–910.
[21] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *ISSTA*. 166–176.
[22] Daniel T. Gillespie. 2007. Stochastic Simulation of Chemical Kinetics. *Annual Review of Physical Chemistry* 58, 1 (2007), 35–55.
[23] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN'82)*. 120–126.
[24] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *ASE*.
[25] J. Hillston. 1996. *A Compositional Approach to Performance Modelling*. Cambridge University Press.
[26] C Hrischuk, J Rolia, and C Murray Woodside. 1995. Automatic generation of a software performance model using an object-oriented prototype. In *MASCOTS*.
[27] Emilio Incerto, Annalisa Napolitano, and Mirco Tribastone. 2018. Moving Horizon Estimation of Service Demands in Queuing Networks. In *MASCOTS*.
[28] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2016. Symbolic Performance Adaptation. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*.
[29] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. 2017. Software Performance Self-Adaptation through Efficient Model Predictive Control. In *ASE*.
[30] Pooyan Jamshidi, Miguel Velez, Christian Kästner, and Norbert Siegmund. 2018. Learning to sample: exploiting similarities across environments to learn performance models for configurable systems. In *ESEC/FSE*.
[31] Amir Kalbasi, Diwakar Krishnamurthy, Jerry Rolia, and Michael Richter. 2011. MODE: Mix driven on-line resource demand estimation. In *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 1–9.
[32] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based sampling of software configuration spaces. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 1084–1094.
[33] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6980
[34] Matthias Kowal, Ina Schaefer, and Mirco Tribastone. 2014. Family-Based Performance Analysis of Variant-Rich Software Systems. In *Fundamental Approaches to Software Engineering (FASE)*. 94–108.
[35] Matthias Kowal, Max Tschaikowski, Mirco Tribastone, and Ina Schaefer. 2015. Scaling Size and Parameter Spaces in Variability-Aware Software Performance Models. In *ASE*. 407–417. https://doi.org/10.1109/ASE.2015.16
[36] Heiko Koziolek. 2010. Performance evaluation of component-based software systems: A survey. *Performance Evalutation* 67, 8 (2010), 634–658.
[37] T. G. Kurtz. 1970. Solutions of ordinary differential equations as limits of pure Markov processes. In *J. Appl. Prob.*, Vol. 7. 49–58.
[38] Marin Litoiu. 2019. Panel: AI and Performance. In *International Conference on Performance Engineering (ICPE)*. https://icpe2019.spec.org/conference-program.html#session5
[39] Zhen Liu, Laura Wynter, Cathy H Xia, and Fan Zhang. 2006. Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation* 63, 1 (2006), 36–60.

[40] Daniel A Menasce. 2008. Computing Missing Service Demand Parameters for Performance Models.. In *Int. CMG Conference*. 241–248.

[41] Tom M. Mitchell. 1997. *Machine learning*. McGraw-Hill. http://www.worldcat.org/oclc/61321007

[42] Object Management Group. 2007. *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE). Beta 1.* OMG. OMG document number ptc/07-08-04.

[43] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. 2008. CPU demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation* 65, 6-7 (2008), 531–553.

[44] Barak A Pearlmutter. 1989. Learning state space trajectories in recurrent neural networks. *Neural Computation* 1, 2 (1989), 263–269.

[45] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. 2017. Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models. *CoRR* abs/1708.08296 (2017).

[46] Abhishek B Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M Voelker. 2008. Automatic request categorization in internet services. *ACM SIGMETRICS Performance Evaluation Review* 36, 2 (2008), 16–25.

[47] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence Models for Highly Configurable Systems. In *ESEC/FSE*.

[48] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *ICSE*. 167–177.

[49] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. *Performance Evaluation* 92 (2015), 51–71.

[50] William J. Stewart. 2007. Performance Modelling and Markov Chains. In *SFM*. 1–33.

[51] Charles Sutton and Michael I Jordan. 2011. Bayesian inference for queueing networks and modeling of Internet services. *The Annals of Applied Statistics* (2011), 254–282.

[52] Alexander Tarvo and Steven P. Reiss. 2014. Automated analysis of multithreaded programs for performance modeling. In *ASE*.

[53] Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.

[54] Mirco Tribastone. 2010. Relating layered queueing networks and process algebra models. In *WOSP/SIPEW*.

[55] Mirco Tribastone. 2013. A Fluid Model for Layered Queueing Networks. *IEEE Transactions on Software Engineering* 39, 6 (2013), 744–756. https://doi.org/10.1109/TSE.2012.66

[56] Mirco Tribastone, Jie Ding, Stephen Gilmore, and Jane Hillston. 2012. Fluid Rewards for a Stochastic Process Algebra. *IEEE Trans. Software Eng.* 38 (2012), 861–874.

[57] Mirco Tribastone, Stephen Gilmore, and Jane Hillston. 2012. Scalable Differential Analysis of Process Algebra Models. *IEEE Transactions on Software Engineering* 38, 1 (2012), 205–219. https://doi.org/10.1109/TSE.2010.82

[58] Mirco Tribastone, Philip Mayer, and Martin Wirsing. 2010. Performance Prediction of Service-Oriented Systems with Layered Queueing Networks. In *Leveraging Applications of Formal Methods, Verification, and Validation (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 6416. Springer, 51–65. http://cse.lab.imtlucca.it/~mirco.tribastone/papers/isola2010.pdf

[59] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring Performance Prediction Models Across Different Hardware Platforms. In *ICPE*.

[60] Weikun Wang and Giuliano Casale. 2013. Bayesian service demand estimation using Gibbs sampling. In *MASCOTS*.

[61] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj Nambiar. 2016. Maximum likelihood estimation of closed queueing network demands from queue length data. In *ICPE*.

[62] Murray Woodside, Greg Franks, and Dorina C Petriu. 2007. The future of software performance engineering. In *Proceedings of the Future of Software Engineering (FOSE)*. 171–187.

[63] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual networks. *arXiv preprint arXiv:1605.07146* (2016).

[64] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic Profiling. In *PLDI*. 67–76.

A APPENDIX

Proof of Theorem 2.1. We construct $\hat{\mathbf{P}}$ and $\hat{\boldsymbol{\mu}}$ as follows:

$$\hat{\mathbf{P}}_{k,i} = \begin{cases} \pi_k & \text{if } i = k \\ \frac{\mathbf{P}_{k,i}}{1-\mathbf{P}_{k,k}}(1-\pi_k) & \text{if } \mathbf{P}_{k,k} < 1 \text{ and } i \neq k \\ \frac{1-\pi_k}{M-1} & \text{otherwise} \end{cases}$$

$$\hat{\boldsymbol{\mu}}_k = \begin{cases} \frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k & \text{if } \mathbf{P}_{k,k} < 1 \\ 0 & \text{otherwise} \end{cases}$$

We prove that, for each $i \neq k$ we have $\hat{\mathbf{P}}_{k,i}\hat{\boldsymbol{\mu}}_k = \mathbf{P}_{k,i}\boldsymbol{\mu}_k$ and $(\hat{\mathbf{P}}_{k,k}-1)\hat{\boldsymbol{\mu}}_k = (\mathbf{P}_{k,k}-1)\boldsymbol{\mu}_k$. Then (a) follows by substitution.

We now consider the case $\mathbf{P}_{k,k} < 1$.

$$\begin{aligned} \hat{\mathbf{P}}_{k,i}\hat{\boldsymbol{\mu}}_k &= \frac{\mathbf{P}_{k,i}}{1-\mathbf{P}_{k,k}}(1-\pi_k)\frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k \\ &= \frac{\mathbf{P}_{k,i}}{\mathbf{P}_{k,k}-1}(\pi_k-1)\frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k \\ &= \mathbf{P}_{k,i}\boldsymbol{\mu}_k. \\ (\hat{\mathbf{P}}_{k,k}-1)\hat{\boldsymbol{\mu}}_k &= (\pi_k-1)\frac{\mathbf{P}_{k,k}-1}{\pi_k-1}\boldsymbol{\mu}_k \\ &= (\mathbf{P}_{k,k}-1)\boldsymbol{\mu}_k. \end{aligned}$$

We now consider the case $\mathbf{P}_{k,k} = 1$. We remark that, in this case, $\mathbf{P}_{k,i} = 0$ if $i \neq k$.

$$\begin{aligned} \hat{\mathbf{P}}_{k,i}\hat{\boldsymbol{\mu}}_k &= \frac{1-\pi_k}{M-1}0 = 0 = 0\boldsymbol{\mu}_k = \mathbf{P}_{k,i}\boldsymbol{\mu}_k. \\ (\hat{\mathbf{P}}_{k,k}-1)\hat{\boldsymbol{\mu}}_k &= (\pi_k-1)0 = 0 = 0\boldsymbol{\mu}_k = (\mathbf{P}_{k,k}-1)\boldsymbol{\mu}_k. \end{aligned}$$

The point (b) is true by definition of $\hat{\mathbf{P}}$. Statement (c) can be shown as follows. When $\mathbf{P}_{k,k} < 1$:

$$\begin{aligned} \sum_i \hat{\mathbf{P}}_{k,i} &= \hat{\mathbf{P}}_{k,k} + \sum_{i\neq k}\hat{\mathbf{P}}_{k,i} \\ &= \pi_k + \sum_{i\neq k}\frac{\mathbf{P}_{k,i}}{1-\mathbf{P}_{k,k}}(1-\pi_k) \\ &= \pi_k + 1 - \pi_k = 1 \end{aligned}$$

where the last statement follows because $\sum_i \mathbf{P}_{k,i} = 1$, $\sum_{i\neq k}\mathbf{P}_{k,i} = 1-\mathbf{P}_{k,k}$. When $\mathbf{P}_{k,k} = 1$:

$$\begin{aligned} \sum_i \hat{\mathbf{P}}_{k,i} &= \hat{\mathbf{P}}_{k,k} + \sum_{i\neq k}\hat{\mathbf{P}}_{k,i} \\ &= \pi_k + \sum_{i\neq k}\frac{1-\pi_k}{M-1} \\ &= \pi_k + \frac{M-1}{M-1}(1-\pi_k) \\ &= \pi_k + 1 - \pi_k = 1 \end{aligned}$$

Statement (d) can be shown observing that $0 \leq \pi_k < 1$, $1-\mathbf{P}_{k,k} \geq 0$ (since $\mathbf{P}_{k,k} \leq 1$) and $1-\pi_k > 0$. Statement (e) can be shown observing that $\boldsymbol{\mu}_k \geq 0$, $\mathbf{P}_{k,k}-1 \leq 0$ and $\pi_k - 1 < 0$. □