ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE
IN
TELECOMMUNICATIONS ENGINEERING

AI-BASED CHARACTERIZATION OF MICROSERVICE RESPONSE TIME IN CLOUD-NATIVE DEPLOYMENTS

Master Thesis
in
Laboratory of Advanced Networking M

Supervisor
Prof. GIANLUCA DAVOLI

Candidate
LORENZO MANCA

Co-supervisor
Dr. DAVIDE BORSATTI

SESSION III
ACADEMIC YEAR 2021/2022

To my family, friends and everyone who has supported me on this journey. Special thanks to my supervisor and co-supervisor for their support and valuable advice.

Contents

Abstract

This thesis addresses the important topic of future networks and new technologies for cloud-native deployment and the exploitation of artificial intelligence (AI) for characterizing microservice response times. In particular, the characterization relies on the use mixture density networks (MDN), a feedforward neural network (FNN) architecture to approximate the distributions of a phenomenon, applied to generalize microservices response time distributions depending on system parameters.

The AI approach is motivated by the difficulty of using theoretical queueing models to characterize the full service. The thesis also explores the possibility of using the distributions obtained through MDN models to make simulations with a digital twin of the system.

The experiments show the approximation effectiveness of this approach for characterizing the response times and to make use of the results in a digital twin.

Finally the thesis makes considerations about possible improvements and future works in this direction.

Chapter 1

Introduction

1.1 Motivations

The advent of 5G technologies has opened the way for new applications and services, but telecommunications research does not stop there. At the same time cloud computing has become one of the most popular and requested technology services in the world, making possible the availability of computing resources on a large scale; this service has become possible thanks to virtualization, which allows computing resources to be created and managed flexibly and efficiently. Today, virtualization through containers has become a fundamental tool to manage and deploy applications in the cloud.

The networks of the future will have several key characteristics, including: higher speed, reliability, and sustainability. In fact, the networks of the future will be able to offer even higher connection speeds than today's networks, which will enable them to support data-intensive applications such as augmented and virtual reality, telemedicine, and more. Networks also will need to be highly reliable, capable of handling large amounts of data and traffic securely. Flexibility will be needed to adapt quickly to market changes and new technologies.

Other key point will be sustainability and security: the networks of the future will need to be sustainable, minimizing environmental impact and using renewable energy sources, and they will need to ensure a high level of security, protecting user data from cyber attacks, hacking and fraud.

With the evolution to 6G technology it is expected an increase in the amount of data and applications that need to be managed and distributed,

bringing to new challenges and opportunities to address.

In this context, the adoption of machine learning and implementation of digital twins could be the key to achieving superior performance and more sophisticated and customized services.

Machine learning can provide predictive models for network performance, while digital twin allows the network to be simulated virtually, providing a secure and cost-effective testing environment.

The integration of these technologies could lead to better network management and performance optimization, thereby increasing the efficiency and the offered quality of services .

This thesis aims to explore the potential of implementing machine learning and digital twin to telecommunications, in order to understand the challenges and opportunities that these technologies can offer and contribute to their implementation and development.

The remainder of this thesis is structured as follows. The next section 1.2 is a brief summary of inspirational articles for this thesis, then chapter 2 presents the main concepts and technologies related to the cloud-native deployment. Chapter 3 introduces the notions concerning artificial intelligence and the machine learning methodologies applied in the experimental part of this thesis while chapter 4 gives a description of the system under analysis and collects the experiments. Finally, chapter 5 summarizes the work and gives suggestions for possible improvements and new approaches to be evaluated.

1.2 Related works

The literature provides a valuable perspective for evaluating the current state of the art and identifying future opportunities.

In particular, [1] accounts for the importance of Kubernetes, an open-source orchestration platform for automatic deployment, scaling and management of containerized applications. Solutions for evaluating the behaviour of the systems with different configuration parameters in Kubernetes are needed to effectively deal with resources scheduling. In this direction [2] argues the performance assessment during the investigation of possible alternative deployments, requiring new service management tools providing *what-if scenario* analysis functions. The use of simulations tools to drive decisions is experimented as a solution to identify the most convenient options.

Finally, [3] underlines the importance of dynamic and intelligent managing of heterogeneous resources such as communication, computing, energy, and storage to improve resource utilization efficiency and satisfy quality of service requirements. Digital twin technology is proposed as a solution to achieve this, indeed many components need to be emulated (e.g., processing nodes and communication network devices) with requirements on latency and accuracy to maintain real-time consistency with physical systems.

Chapter 2

Overview

This chapter presents the main concepts and technologies related to cloud-native computing, such as virtualization, virtualization with containers and service composition.

2.1 Cloud-native deployment

Cloud-native service deployment refers to the practice of deploying applications and services within a cloud environment using architectures and methodologies that take advantage of cloud computing features. Specifically, cloud-native deployment focuses on creating highly scalable, reliable and flexible services that can be easily managed and updated in a cloud environment.

This requires the adoption of a number of specific development and management practices, including the use of microservices, service-oriented design, virtualization, containerization, automation, error handling and self-healing. In addition, the use of cloud-native technologies and platforms, such as Kubernetes, enables greater agility and scalability, as well as the ability to deliver services quickly and reliably.

The concept of *DevOps* is strictly related to the cloud-native context. Indeed DevOps is an organizational and development culture that involves collaboration between development and operations teams to improve the efficiency and quality of application development and deployment processes. DevOps focuses on continuous integration (CI) and continuous release (CR) to ensure the rapid and reliable release of applications.

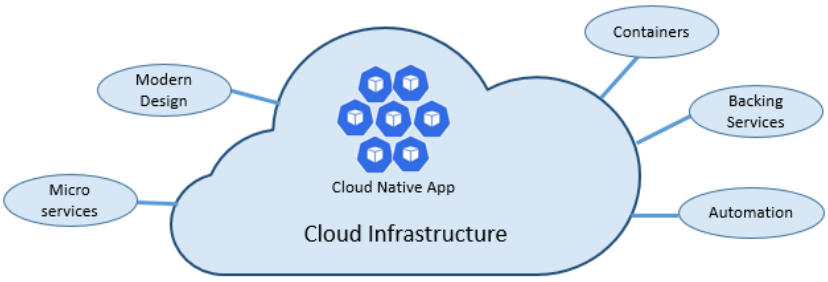


Figure 2.1: Cloud-native pillars. [4]

2.2 Virtualization

The virtualization is a technique aiming to create abstract versions of computer hardware and network resources to make them available to a software as *virtual resources*. This technique allows the implementation of *virtual machines* that act like a real computer. As a matter of fact, a virtual machine is equipped with its operating system and applications. The virtualization of hardware requires an *hypervisor*, a software which manages the physical resources and isolates them from the virtual environment; this task is reached by sharing the resources.

In traditional virtualization there are two kind of hypervisors:

- Type 1 (native): in this paradigm the hypervisor directly schedules the virtual machines (VM) resources to the hardware, no OS is present on the host machine. An example of this kind of hypervisor is the KVM (Kernel-based virtual machine).
- Type 2 (hosted): in this paradigm the hypervisor is a process executing on a OS.

The benefits introduced by the use of the virtualization are:

- Cost reduction: virtualization allows to reduce the number of physical computer by executing many virtual machines on a single physical one. This reduces the capital expenditures, and the operational costs related to cooling and maintenance.
- Flexibility: any virtual machine can be executed on any available server

on the network, it is possible to move a VM from a server to another in case of need. Moreover, it is possible to scale VM quickly.

- Reliability: it is possible to make checkpoints and backups of VM in such a way any VM can be easily restored. This makes virtual system fault resistant.

- Control independence and DevOps: developers can easily start a VM without having an impact on the production environment since in virtualization the VMs are independent.

2.3 Virtualization with containers

In the virtualization context, the use of *containers* is a effective technique to manage applications and services in a shared environment. The definition of container can be summarized in "a single executable package of software". Differently to traditional virtualization, where every VMs have their own OS, the use of containers is based on the host kernel to run applications and so no hypervisor is needed. To make containers work, the host kernel has installed a runtime engine to enable containers to interact with it [5].

The use of host kernel makes the containers "lightweight", meaning the image of a container is typically in the order of megabyte, much smaller than the image of a VM that is in the order of gigabyte. Moreover a container does not embed anything bigger than an application and its running environment, for this reason they are often used to implement only a single function performing specific task, known as *microservice*.

The "lightweight" architecture of containers enables the possibility to easily move and run an application in any environment or infrastructure's operating system. This property turns out to be very useful to deploy cloud-native apps (i.e., collection of microservices whose composition provides a more complete service).

An example of containerization technology is *Docker*, enabling the creation and use of Linux containers.

The virtualization with containers extends the benefits introduced by the traditional virtualization, with:

- Portability: possibility to run uniformly and consistently across any platform or cloud being abstracted away from the host operating system;

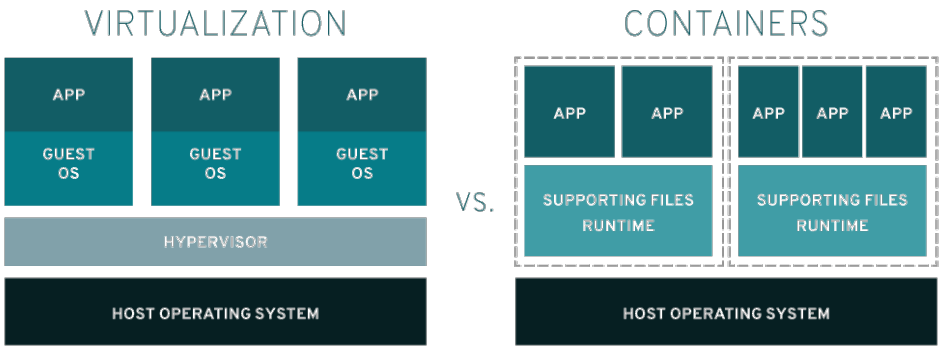


Figure 2.2: Virtualization vs. Containerization. The structure of container is simpler and requires less overhead. [6]

- Speed and efficiency: the lightweight property of containers makes the start-up faster;
- Easy management: a container orchestration platform can automate the installation and handling of containers' workloads and services (e.g., rolling out new versions of applications, provide logging and debugging).

2.3.1 Microservices and service composition

The microservice architecture is an approach to break a complex application into a collection of many specialized services, called *microservices*.

The microservice architecture is opposed to the monolithic application approach where components such as user interface (UI), business logic and data access functions are implemented in a single software application responsible for every step needed to complete a service.

The main benefit introduced by the in microservice architecture is the possibility to focus the development phase on a specific task, without having an impact on the whole service, making the development, test and deployment of applications faster.

Microservices works well if combined with containerization, since they can inherit all the benefits of virtualization with containers.

A complete service is obtained by the composition of multiple microservices, this objective is obtained through Application Program Interfaces (APIs) and REST interfaces (such as HTTP).

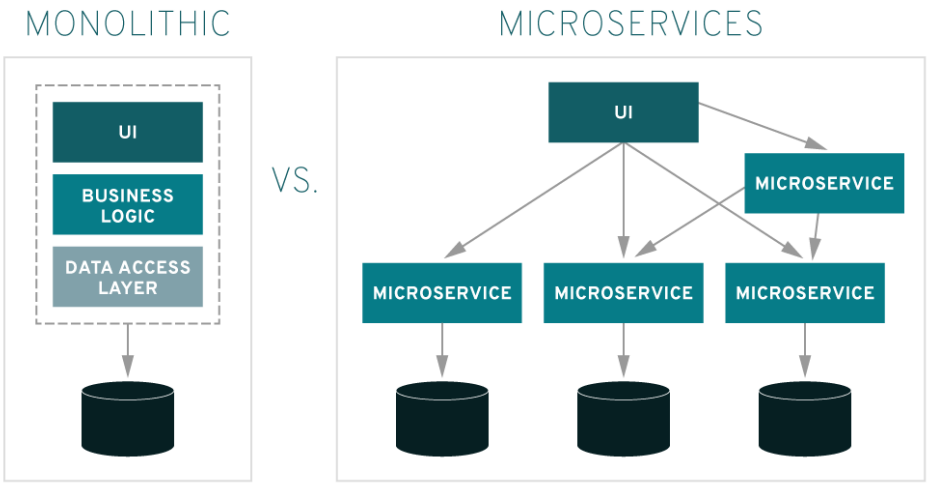


Figure 2.3: Monolithic vs microservices.[6]

In case of large compositions it is possible to adopt a *service mesh*, which is a facility dedicated to the service-to-service communication between services or microservices using proxies: when microservices needs to communicate the requests are routed between them through proxies that are integrated in their own infrastructure layer.

The service mesh captures every aspect of the inter-service communications, without it the communication needs to be defined with specific rules in the implementation of the microservice itself, making the management more difficult.

2.3.2 Kubernetes

In the scope of container orchestration, *Kubernetes*, known also as k8s, is a popular open source framework that automates container-related functions[7]. It works with common containers engines, such as Docker, to schedule and automate the deployment, management and scaling of containerized applications.

The Kubernetes software is made up by multiple components, including:

- Working nodes: they are working machines on which containers are executed.

- Clusters: it is a set of working nodes;
- Control plane components: they make decisions about the cluster, detect and respond to cluster event;
- Node components: they are the components running on each working node, maintaining running pods (components of the application workload) and enabling the Kubernetes running environment.

Kubernetes features are:

- Service discovery;
- Load balancing;
- Automated rollouts and rollbacks;
- Self healing;
- Secret and configuration management.

2.4 6G

The Sixth Generation (6G) of mobile networks is currently at the conception phase, this new technology is expected to improve the 5G networks with[8, 9]:

- Higher throughput;
- Ultra low-latency;
- Advanced security;
- Environmental sustainability.

In the context of cloud-native, it is expected that 6G will offer new advanced applications thanks to network function virtualization and service orchestration. In particular, the virtualization with containers and the microservice architecture introduce new opportunities to the development and provisioning of applications on 6G networks.

In addition, it is expected that 6G will make use of artificial intelligence (AI) to improve the efficiency of the network, for instance by optimizing network resource usage, implementing new smart beamforming techniques and customize the user experience based on its preferences.

Chapter 3

Methodologies

In this chapter the methodologies applied in the experimental part of the thesis are introduced. Before dealing with model theory, some general concepts related to artificial intelligence and machine learning are covered.

3.1 Artificial intelligence and machine learning

The artificial intelligence (AI) refers to the ability to design techniques that enable machines to mimic human intelligence, meaning the ability to perceiving and inferring information.

Machine learning (ML) is a field of studies of artificial intelligence, it exploits statistical methods to give computers the ability to learn without being explicitly programmed.

ML algorithms are divided in different categories:

- Supervised learning;
- Unsupervised learning;
- Reinforcement learning.

Supervised machine learning is referred to algorithms that maps an input x on an *output label* y. It learns from being given "right answer" examples. Applications of this category include speech recognition, translation, advertising, self-driving cars etc...

The most common algorithms of this class are:

- Regression, aiming to predict a number among infinitely many possible outputs;
- Classification, aiming to predict a class/category among a small number of possible outputs (classes/categories are not real number, they belong to a finite set).

Unsupervised machine learning on the other hand works on data which is not associated to any output label, this kind of ML aims to *find hidden structures in the data*. Given some input data, the algorithm can group them in different clusters (clustering applications, like google news, DNA microarray types, marketing segmentation).

Examples of unsupervised ML algorithms are:

- Clustering, aiming to group similar data points together (e.g., Google News, DNA microarray types, market segmentation);
- Anomaly detection, aiming to find unusual data inputs/events;

3.2 Neural networks and deep learning

Artificial neural networks (NN) are models to perform machine learning. The name "neural network" derives from the fact this architecture is loosely inspired on mammal's brain neural networks.

The architecture of NN is composed by a collection of processing nodes (or *units*) that are densely interconnected. Units are organized in layers, whose configuration provide a specific transformation of the input data. Data travel from the first layer, the *input layer*, to the last one, the *output layer*, passing through the intermediate layers. Data travelling in the NN can have different types (e.g., real numbers or categorical data) and can be *labelled* (i.e., meaning of data is known) or *unlabelled* (i.e., meaning of data is unknown). Units of a layer are interconnected with the ones of the subsequent layer by means of *edges*. Each edge is characterized by a weight, a coefficient that is multiplied to the data passing through it and increasing or decreasing the impact of that data on the final output. All units are characterized by an *activation function* and a *bias*.

The first and simplest model of NN representing a single unit is the *Perceptron*, demonstrated by Frank Rosenblatt in 1957. The aim of the Perceptron was to perform binary classification.

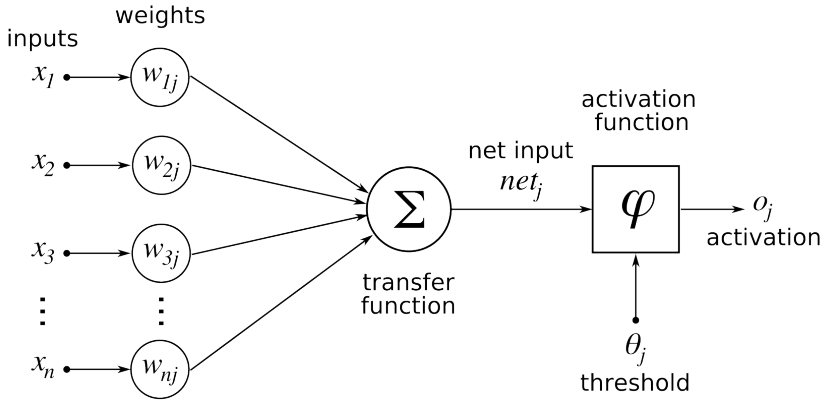


Figure 3.1: Artificial neuron.

Figure 3.1 shows a network with a single hidden layer and one unit. The output of the generic "j" unit is related to the inputs by the relationship:

$$o_j = \phi(\theta_j + \sum_{i=1}^{n} w_{ij} x_i)$$

Where:

- x_i is the i-th input;
- w_{ij} is the weight of the i-th input edge;
- θ_j is the bias;
- $\phi(\cdot)$ is the activation function.

Common choices for the activation function are:

- Logistic;
- Tanh;
- Sigmoid;

- Step;
- Linear;
- Rectified linear unit.

The NN composed by multiple intermediate units and layers are known as *Deep Neural Networks* (DNN) and the algorithm to train them is usually referred as to *Deep learning*. The layers placed between the input layer and the output layer are called *hidden layers* since their results can't be seen directly. Figure 3.2 represent a DNN.

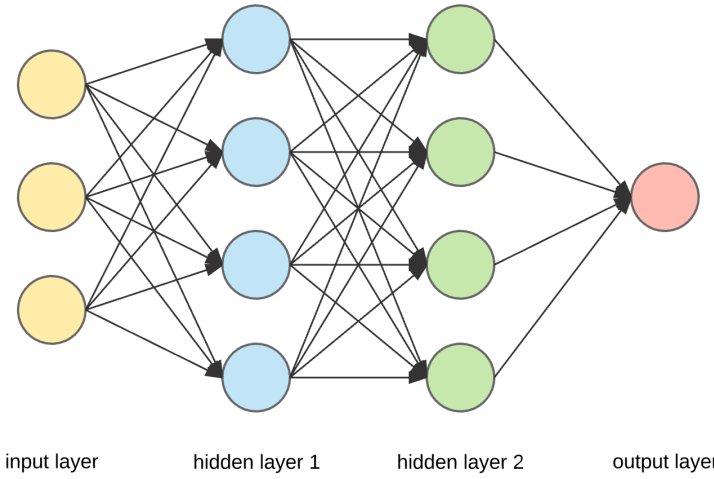


Figure 3.2: DNN with 2 hidden layer of 4 units each, 3 units input and 1 unit output.

3.2.1 Feedforward neural networks

Feedforward neural network (FNN) is an architecture of artificial neural network where connections between the nodes do not form a cycle, this kind of networks are the basis for most complex networks architecture . It is a general model to approximate non-linear mappings between two sets of variables [10]:

$$\hat{y} = f(\bar{x}) \tag{3.1}$$

Where:

- $\bar{x}$ is a vector of inputs (or *features*);
- $\hat{y}$ is the prediction for y (the target).

This model can be extended to multivariate prediction by implementing multiple output neurons. FNN can have multiple hidden layers; the number of hidden layers and the approximation capability concern the *Universal approximation theorem* [11]: a neural network with suitable number of hidden layers and units is a "Universal approximator".

The training of a NN is the process of determining the values of the weights that minimize the error in the approximation. The training is performed on a training set (a set of examples, previously collected, on which the task is learned); feedforward NN are usually trained with the *Back-propagation algorithm*. The error of a neural network is provided by the *error function* (sometimes called cost function or loss), which is specific to the task to be approximated. Once the network is trained, the network is able to process new data.

Feedforward NN are suitable models to solve regression and classification problems. In the regression case the task is learned by minimizing the *mean-square error function* between the target and the prediction given features over the training set.

The mean-square error function is defined as:

$$E(\bar{w}) = \frac{1}{2m} \sum_{i=1}^{m} [f(\bar{x}^{(i)}; \bar{w}) - t^{(i)}]^2 \tag{3.2}$$

Where:

- $\bar{x}^{(i)}$ is the i-th input vector in the training set;
- m is the cardinality of the training set;
- $t^{(i)}$ is the i-th target in the training set;
- $f(\bar{x}^{(i)}; \bar{w})$ is the prediction of the target in function of the i-th input and weights $\bar{w}$

The larger is the training set, the less is the bias and variance of the model. In the limit case of infinite training set the mean-squared error function turns out to be:

$$E = \lim_{m\to\infty} \frac{1}{2m} \sum_{i=1}^{m} [f(\bar{x}^{(i)}; \bar{w}) - t^{(i)}]^2 = \iint [f(\bar{x}; \bar{w}) - t]^2 p(t, \bar{x}) dt d\bar{x} \tag{3.3}$$

The error can be minimized in function of $f(\bar{x}; \bar{w})$ by differentiation:

$$\frac{\partial E}{\partial f(\bar{x}; \bar{w})} = 0 \tag{3.4}$$

By substituting 3.3 in 3.4 and remembering $p(t, \bar{x}) = p(t|\bar{x})p(\bar{x})$ the approximating function that minimize the error (i.e., the optimal estimator) is:

$$f(\bar{x}; \bar{w}) = \mathbb{E}[t|\bar{x}] \tag{3.5}$$

This means the network approximates the conditional average of the target (conditioned to the input vector).

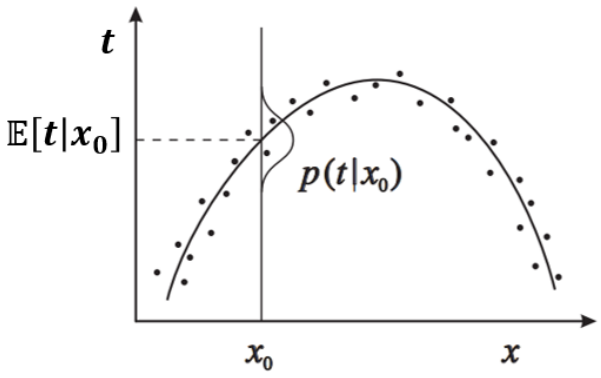


Figure 3.3: The MSE estimator function on a set of training points.[12]

In the case of classification, the task is learned by minimizing the cross-entropy error function and the approximated average gives the posterior probability of a class conditioned on the input features.

3.2.2 Mixture density networks

Feedforward conventional neural networks for regression (prediction of a continuous variable) sometimes provide a limited description of the approximation of the target value, the conditional average of several correct values is not necessarily the correct prediction of a target but it only minimizes the sum-of-squared error function. To have a more complete description of the target, in particular for target with high variability, it is possible to model the *conditional probability distribution* of the target data, given the input vector.

In order to model the conditional probability of the target, a model combining a conventional FNN and a mixture density model can be adopted. This model is called *Mixture Density Network (MDN)* [12] and provides a general framework to approximate conditional density functions of targets by modelling the probability parameters as functions of the features.

In case the target follows a Normal distribution the parameters of its distribution would be (μ, σ), it is possible to write:

$$p(t|\bar{x}) = p(t|\mu(\bar{x}), \sigma(\bar{x})) \tag{3.6}$$

The MDN aim to model the conditional probability density as a linear combination (*mixture model*) of kernel functions:

$$p(t|\bar{x}) = \sum_{c=1}^{C} \alpha_c(\bar{x}) \phi_c(t|\bar{x}) \tag{3.7}$$

Where:

- $p(\bullet)$ is the probability density function (pdf) of the target variable;
- C is the number of kernel components in the mixture;
- $\alpha_c(\bar{x})$ is the mixing coefficient of the c-th kernel, it represents the prior probability of that kernel conditioned on $\bar{x}$;
- $\phi_c(t|\bar{x})$ is the c-th kernel, it represents the conditional density function of the target t.

In the case of Normal kernel it is:

$$\phi_c(t|\bar{x}) = \frac{1}{\sqrt{2\pi}\sigma(\bar{x})} \exp\left\{ \frac{|t - \mu_c(\bar{x})|^2}{2\sigma_c^2(\bar{x})} \right\} \tag{3.8}$$

Each kernel component is then characterized by a set of three parameters dependent on $\bar{x}$: $\big(\alpha_c(\bar{x}), \mu_c(\bar{x}), \sigma_c(\bar{x})\big)$.

The assumption of Normal kernel allows to approximate every probability density function with arbitrary accuracy [12]; this property is particularly convenient in case of multivariate target where the hypothesis of statistical independent component doesn't hold. However, the model can be generalized to different kernel choices (e.g., Exponential, Gamma or other distributions).

The mixture parameters, namely the mixing coefficients $\alpha_c(\bar{x})$, the means $\mu_c(\bar{x})$ and the standard deviations $\sigma_c(\bar{x})$ are assumed to be unknown continuous functions of $\bar{x}$.

In order to model the unknown functions:

- the first stage of the model is a *conventional FNN*, with input $\bar{x}$, weights $\bar{w}$ and providing a parameters' vector $\bar{y}(\bar{x}; \bar{w})$ as output (3.1);
- the output vector of the first stage is provided as input of a second stage represented by a *mixture model*.

In the mixture model the input parameters $\bar{y}(\bar{x}; \bar{w})$ are processed by means of appropriate activation functions to model the mixture parameters (i.e., mixing coefficients, means and std deviations). The final output of the model turns out to be the conditional probability density (3.7). The combination of the two stages is referred to as a *Mixture Density Network* (MDN).The basic structure is represented in figure 3.4

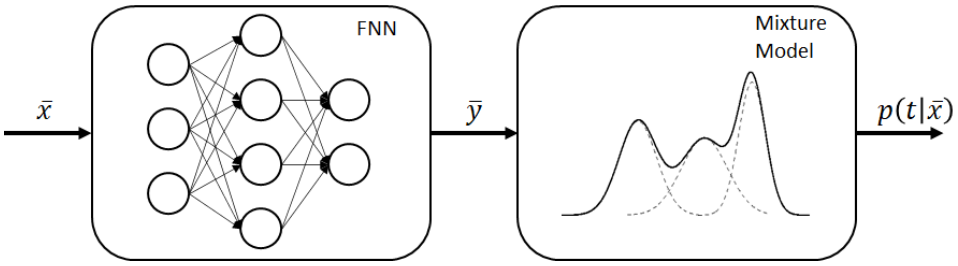


Figure 3.4: Structure of a MDN

The adoption of this model requires the definition of some hyper-parameters (i.e., fixed parameters defining the specific implementation of the NN, they are

not learned during training) such as the number of mixture components C, the kernel functions and the number of hidden layers and units.

The mixing coefficients $\alpha_c(\bar{x})$ must satisfy the condition:

$$\sum_{c=1}^{C} \alpha_c(\bar{x}) = 1 \tag{3.9}$$

It can be achieved by choosing the *softmax* activation function for these outputs and the generic i-th mixing coefficient is:

$$\alpha_i = \frac{\exp(y_{\alpha_i})}{\sum_{c=1}^{C} \exp(y_{\alpha_c})} \tag{3.10}$$

where y_{α_i} represents the FNN output related to the i-th mixing coefficient.

The variances $\sigma_c(\bar{x})$ must satisfy the condition:

$$\sigma_c(\bar{x}) > 0 \tag{3.11}$$

It can be achieved by using *exponential* activation function:

$$\sigma_i = \exp(y_{\sigma_i}) \tag{3.12}$$

where y_{σ_i} represents the FNN output related to the i-th variance.
The mean values $\mu_c(\bar{x})$ are not constrained by any condition, they can be obtained by the FNN output with linear activation function:

$$\mu_i = y_{\mu_i} \tag{3.13}$$

The parameters $\bar{w}$ of $\bar{y}(\bar{x}; \bar{w})$ are learned during training of the MDN through an examples set $\left\{\bar{x}^{(q)}, t^{(q)}\right\}$ of cardinality m. This can be obtained by maximum likelihood estimation (MLE), indeed the objective of MLE is to find the set of parameters for which the observed data (the training set) have the highest joint probability. By assuming the training examples are drawn independently from the pdf given by 3.8, the likelihood function of the set can be written as:

$$\mathcal{L} = \prod_{q=1}^{m} p(t^{(q)}, \bar{x}^{(q)}) = \prod_{q=1}^{m} p(t^{(q)}|\bar{x}^{(q)}) p(\bar{x}^{(q)}) \tag{3.14}$$

The maximum likelihood estimate is:

$$\widehat{\bar{w}} = \arg\max_{\bar{w}} \mathcal{L}(\bar{w}) \tag{3.15}$$

From the likelihood it is possible to derive the error function:

$$E = -\log \mathcal{L}(\bar{w}) = -\sum_{q=1}^{m} \log p(t|\bar{x}^{(q)}) \tag{3.16}$$

The 3.16 is called negative log-likelihood (NLL), in its expression the $p(\bar{x})$ factor is neglected since it does not depend on the NLL parameters. Its minimization is equivalent to the maximization of the likelihood.

By taking into account the mixture model 3.7, the NLL becomes:

$$E = \sum_{q=1}^{m} E^{(q)} = -\sum_{q=1}^{m} \log \sum_{c=1}^{C} \alpha_c(\bar{x}^{(i)}) \phi_c(t^{(i)}|\bar{x}^{(i)}) \tag{3.17}$$

where $E^{(q)} = -\log \sum_{c=1}^{C} \alpha_c(\bar{x}^{(q)}) \phi_c(t^{(q)}|\bar{x}^{(i)})$ is the error contribution of the q-th example.

Back-propagation procedure is used to minimize the error function, for this purpose the gradient of the error function with respect to the FNN output needs to be computed as reported in [10]. The software implementation of the back-propagation algorithm for this model can be inspired on regression FNN based on sum-of-squares error function. The modification of the error function is required to apply standard optimization procedures such as gradient descent.

The derivative of the error on the q-th example with respect to a generic weight w is:

$$\frac{\partial E^{(q)}}{\partial w} = \sum_i \frac{\partial E^{(q)}}{\partial y_i} \frac{\partial y_i}{\partial w} = \sum_i \delta_i^{(q)} \frac{\partial y_i}{\partial w} \tag{3.18}$$

where $\delta_i^{(q)} = \dfrac{\partial E^{(q)}}{\partial y_i}$ is the derivative of the error with respect to the i-th FNN output that is back-propagated in the network. The partial derivative can be approximated by finite difference

$$\frac{E(w+\epsilon) - E(w)}{\epsilon} = \frac{\partial E}{\partial w} + \mathcal{O}(\epsilon) \tag{3.19}$$

At the end of the training, the MDN is able to approximate the conditional density function of the target data given the input feature, allowing to have a probabilistic description of the data generation process. The MDN results are useful to derive the specific statistics. By exploiting the 3.7 and linearity the conditional mean is given by

$$\mathbb{E}[t|\bar{x}] = \sum_{c=1}^{C} \alpha_c(\bar{x}) \int t\, \phi_c(t|\bar{x})\, dt = \sum_{c=1}^{C} \alpha_c(\bar{x})\, \mu_c(\bar{x}) \tag{3.20}$$

and the conditional variance

$$\mathrm{Var}(t|\bar{x}) = \mathbb{E}\big[(t - \mathbb{E}[t|\bar{x}])^2 \,|\, \bar{x}\big] \tag{3.21}$$

$$= \sum_{c=1}^{C} \alpha_c(\bar{x}) \left\{ \sigma_c^2(\bar{x}) + \big[\mu_c(\bar{x}) - \sum_{j=1}^{C} \alpha_j(\bar{x})\, \mu_j(\bar{x})\big]^2 \right\} \tag{3.22}$$

3.3 Goodness of fit

The choice of a suitable model among several possibilities needs to be accounting its performance with a metric. The *goodness of fit* of a statistical model is a measure of how well a model fits data, typically this measure allows a data analyst to summarize the discrepancy between a sample of observed values and the predictions made by the model under evaluation. Examples of goodness of fit are Kolmogorov-Smirnov test (KS-test) and Bayesian information criterion (BIC).

3.3.1 Kolmogorov-Smirnov test

Kolmogorov-Smirnov test is a non-parametric test used to compare a sample with a reference probability distribution (one-sample K-S test), or to compare two samples (two-sample K-S test)[13, 14]. The one-sample K-S test aims to quantify the distance between the empirical cumulative distribution function (eCDF) of a sample and the cumulative distribution function (CDF) of a reference distribution.

For a random variable X with CDF $F(x)$ and a sample $(X_1, X_2, ..., X_n)$ with eCDF $F_n(x)$ the test's hypothesis are:

$$H_0 : F(x) = F_n(x), \ \forall x \tag{3.23}$$
$$H_1 : F(x) \neq F_n(x), \ \text{for some } x \tag{3.24}$$

where the eCDF is defined as:

$$F_n(x) = \frac{\text{number of (elements in the sample} \leq x)}{n} \tag{3.25}$$

The test's statistic (K-S distance) is defined as:

$$D_n = \sup_x |F_n(x) - F(x)| \tag{3.26}$$

Under the null hypothesis, the K-S distance (3.26) is distributed according to the Kolmogorov distribution and converges to 0 in case $n \to \infty$. For a large sample ($n > 50$) goodness of fit test is made by setting a critical value K_α on the Kolmogorov distribution and the null hypothesis is rejected at significance level α if:

$$D_n > \frac{K_\alpha}{\sqrt{n}} \tag{3.27}$$

It is possible to perform the test with different variants of the null hypothesis, as implemented in many software libraries.

3.4 Overfitting, underfitting and regularization

The *overfitting* is a problem coming out in machine learning when a model becomes too specific to the training set, this leads to a model which seems very accurate at the end of the training phase (low value of the error function), but actually it shows a lack of generality being unable to fit new data. It is used to say the model is characterized by a *high variance* since slight changes on the training set alter the model very much.

The *underfitting*, as opposite to overfitting, is a problem coming out when the model is not able to fit well the training set, showing a high value of the cost function. It is used to say the model is characterized by a *high bias*.

In general underfitting happens when the model is too simple and it is not able to extract the framework inside the training data, reflecting the same

issue on the new data. For instance, underfitting happens when a model has too few parameters. Overfitting happens when the model is too complex, for instance, it occurs when the model has too many parameters.

Techniques to overcome overfitting and underfitting includes:

- Collection of more training examples: it reduce the variance but sometimes it is not practicable because collect more data could be expensive.
- Feature selection: when the training set includes few examples n but a higher number of features $m > n$ the overfitting may arise if all the features are accounted in the model. In order to avoid overfitting it is possible to select only the most relevant features (i.e., the ones showing higher correlation with the target). The main disadvantage of this technique is that possible important information may be lost.
- Regularization: in case a high number of features are included in the training set, the regularization technique aims to shrink the parameters related to the features that specialize too much the model, reducing their effect without completely eliminating it.

3.5 AI model's development environment

The experiments in the following parts of this thesis were developed by using different tools available in the Python language environment.

Python is a high-level object-oriented programming (OOP) language. The open-source, general-purpose and simplicity properties of this language are the strengths which have ensured its diffusion in many scientific fields, such as data science and machine learning. The Python's tools applied in the thesis belongs to different libraries such as:

- NumPy for scientific computing [15];
- Scipy for statistical functions [16];
- Matplotlib for graphical visualizations [17];
- Pandas for data manipulation [18];

- TensorFlow for machine learning [19].

In the work the major contribution is given by TensorFlow. It is an open source framework developed by Google for building machine learning models, such as neural networks, decision trees, and regression tasks [19]. This tool offers many functionalities for the training and validation of models as well as instruments for the elaboration of results.

In TensorFlow deep networks are developed with the *Keras* API. The first step in the development requires the definition of the structure (i.e., number of layers, neurons and activation functions). Then it is necessary to make available data (i.e., import data in a "dataframe" structure) defining features and targets and performing features normalization to optimize the convergence of the model during training. When data is available, it is necessary to divide the dataset in three disjointed groups:

- Training set: the portion of the dataset used to train the model;
- Validation set: the portion of the dataset used to check the absence of overfitting during training;
- Test set: the portion of the dataset used after the training to check the generalization performance.

When results are available, they can be elaborated and summarized with charts and tables. The results of the thesis work are manipulated with the help of TensorFlow Probability, a library containing probabilistic tools that can be integrated in the models.

Chapter 4

Experiments

This chapter aim to show the application of methodologies described in the 3 on a real system. A description of the real system and its properties is provided to clarify the context of the work.

4.1 Cloud-native deployment scenario under analysis

4.1.1 System description

The cloud-native deployment scenario under analysis in the experimental work of this thesis is represented by means of a *multi-layered queue system*. As a matter of fact, cloud service provisioning can be efficiently implemented using a multi-layered system approach.

The general implementation of a multi-layered queue system can be described by:

- *requests*: the system is fed with incoming requests sent by users which can be summarized by their average request rate $\lambda[req/s]$.
- *replicas*: they are multiple processing entities providing a *microservice* (i.e., an independent portion of processes related to a more complete service); as they are able to operate in parallel, a section with n_{rep} replicas can contemporary execute up to n_{rep} instances of microservices. The processing time related to a microservice performed by a replica is a random variable T_{ms} with unknown distribution.

- *queues*: each replica in the system has its own queue in which requests are queued in case they can't be immediately served.
- *load balancer*: it distributes the incoming requests to the replicas in case many replicas are present; the simplest way to do it is performing equal balancing.

In a cloud-native deployment a service is obtained through the chaining of several microservices, this composition is strictly related on the kind of service provided to the users.

With reference to a particular service, the structure described above can be stacked as many times as the number of microservice that make up the overall service, where each layer is responsible for a specific microservice. Figure 4.1 shows an example.

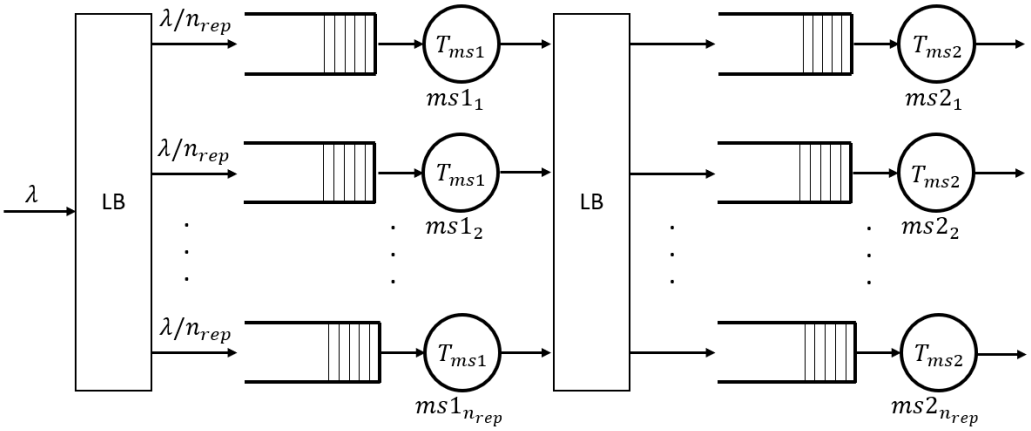


Figure 4.1: Example: a two-layered queue system with n_{rep} replicas for both layers. In the first layer, on average, the load balancer (LB) equally distributes requests among replicas.

The multi-layered queue system during its operations (e.g., load balancing, queuing, microservices processing) introduces a random response time to the users, this is referred as to TTR (time-to-resolution). The distribution of the time-to-resolution is unknown and it depends on the kind of service and the structure of the system.

The specific implementation of the system under analysis in the experiments' part is given by a two-layered queue system with different values of

$n_{rep} \in \{1, 2, 3, 4\}$. The system is made up by using the *Kubernetes* platform and the replicas are deployed by means of *containers* distributed on a cluster.

The designated service for that system is an image processing application, in particular:

- the first layers is dedicated to an image pre-processing microservice (ms1) to simplify the operations of the second microservice;
- the second layer is dedicated to the actual image processing microservice (ms2).

The random processing times of a generic replica in the first and second layers are indicated as T_{ms1} and T_{ms2} respectively.

4.1.2 System data collection

The described system has been tested on different conditions to collect data about time-to-resolutions and microservices processing times. The data collection is fundamental to characterize the system and aim to model its target quantities exploiting machine learning.

Data were collected performing multiple experiments on the system with:

- different number of replicas $n_{rep} \in \{1, 2, 3, 4\}$
- for each number of replicas were produced $n = 5000$ incoming requests to the system with average request rate $\lambda \in \{1, 2, \dots, 30 * n_{rep}\}[req/s]$ in order to compare cases with different number of replicas but with the same average request rate per replica.

At the end of the experiment 300 data sets of 5000 samples were obtained for every target.

4.2 Motivations of the AI-based approach

The multi-layered queue system modelling and, in particular, the characterization of the response times can be done with two different approaches: adoption of theoretical-analytical queue modelling or data-based modelling [20].

The analytical approach aims to model the system exploiting queuing theory, which is a set of mathematical results that can be applied to the components of the system to obtain a characterization of its behavior. This approach is still an open field of research in telecommunications engineering; it is suitable to study simplified cases (e.g., stationary conditions and exponential service time), but can become hard to deal with in the study of more complex systems. The limitation of this approach justify the interest in data-based methods such as machine learning algorithms and the exploitation of simulations.

4.3 Models

In this section the models for the characterization of the system described in the section 4.1.1 with the application of the methodologies described the Chapter 3 are presented.

4.3.1 Model 1: characterization of microservice ms1 response time

The first proposed model aim to characterize by means of a mixture density network (MDN) the microservise ms1 response time, given the knowledge of the request rate to the system and the number of available replicas.

Mathematically the objective of the model is to approximate:

$$p(t_{ms1}|\lambda, n_{rep}) \tag{4.1}$$

Where:

- $p(\bullet)$ indicates the probability density function (pdf);
- $t_{ms1}[s]$ is the ms1 response time;
- (λ, n_{rep}) are respectively the request rate $[req/s]$ and number of available replicas for ms1.

The approximation given by the model is useful to generalize the behaviour of the replicas in the first layer, perform analysis and exploit the pdf to make simulations.

The model is a DNN implemented with a Keras sequential model and TensorFlow probability, with the following characteristics:

- 2 input neurons: one input neuron for λ and one for n_{rep};
- 4 mixture components: the approximated pdf is the superposition of 4 Gamma's pdfs;
- 12 output neurons: the outputs represent the parameters of the components in the mixture model, every Gamma's pdf is characterized by 3 parameters (the weight in the superposition, the concentration and the rate)
- 2 hidden layer: each hidden layer has 7 neurons.

The hidden neurons are set with $ReLU$ (Rectified Linear Unit) activation function. The activation functions for the output neurons depend on the parameter they represent.

The Gamma pdf is:

$$\phi(x, a, b) = \frac{x^{a-1} e^{-bx} b^a}{\Gamma(a)} \tag{4.2}$$

where $a > 0$ is the concentration and $b > 0$ is the rate.

During initial trials the Gamma and Normal pdfs were tried. The Gamma pdf was chosen because it provided the best fit among the two, this best fit can be explained by the fact that the Gamma distribution generalizes the Exponential distribution, often used to model times. In particular the Gamma distribution has support on the positive time axis, instead the Normal distribution has support on the whole real axis and so its tails considers also negative times that can't exists

At the end of the training process the ms1 response time is characterized in the following way:

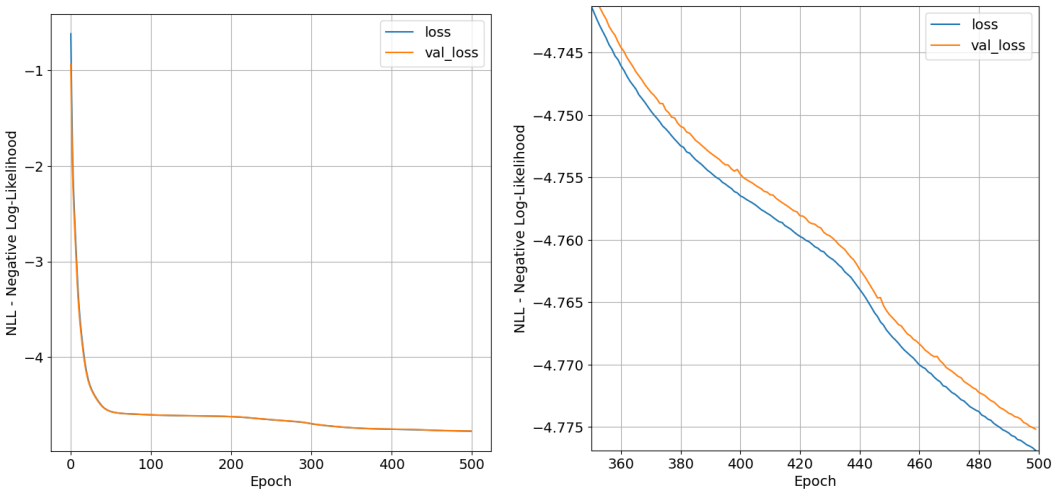
$$p(t_{ms1}|\lambda, n_{rep}) = \sum_{c=1}^{4} \alpha_c(\lambda, n_{rep}) \phi\big(t_{ms1}, a_c(\lambda, n_{rep}), b_c(\lambda, n_{rep})\big) \tag{4.3}$$

```
InputLayer = Input(shape=(2,))  # 2 input neurons

Norm_layer = Normalization(axis = -1)
Layer_1 = Dense(7,activation="ReLU")(Norm_layer(InputLayer))
Layer_2 = Dense(7,activation="ReLU")(Layer_1)
alpha = Dense(4, activation="softmax")(Layer_2)
concentration = Dense(4, activation=lambda x: tf.nn.elu(x) + 1 + 1
    e-15)(Layer_2) # The constant term 1e-15 avoids null values
    that leads to 'NaN' loss
scale = Dense(4, activation=lambda x: tf.nn.elu(x) + 1 + 1e-15)(
    Layer_2)
y_real = Input(shape=(1,))
rate =1/scale
lossF = gammanll_loss(y_real, alpha, concentration, rate)
mdn_ms1_model = Model(inputs=[InputLayer, y_real], outputs=[alpha,
     concentration, rate])
mdn_ms1_model.add_loss(lossF)
# Learning rate scheduler with keras.optimizers.schedules
lr_schedule =tf.keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=i_lr, end_learning_rate=e_lr,
        decay_steps=10000
)
adamOptimizer = optimizers.Adam(learning_rate=lr_schedule)
mdn_ms1_model.summary()
mdn_ms1_model.compile(optimizer=adamOptimizer)
```

Listing 4.1: MDN model implementation

The neurons related to the weights α_c are set with "Softmax" activation functions, the neurons related to the concentrations and the rates are set both with "Exponential" activation functions. Before the training, the dataset was shuffled and split in the following way:

- 70% is dedicated to the training;
- 20% is dedicated to the validation;
- 10% is dedicated to testing.



(a) Training history

(b) Zoom on the last 150 epochs

Figure 4.2: Training history of model 1. The final error is NLL=-4.775 on the validation set

As reported in figure 4.2a the NLL converged after 500 epochs of training, with a batch size of 512 samples. The procedure required approx. 30 min on a laptop with Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz and 8GB RAM (GPU acceleration not exploited).

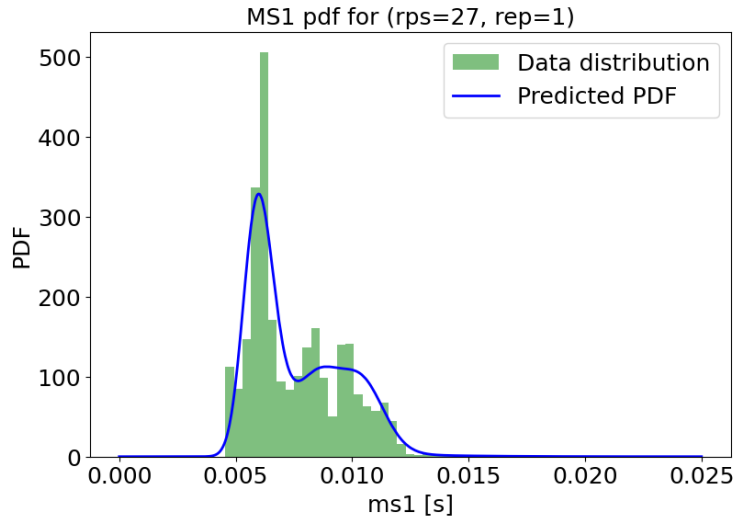
The error function is the Negative Log-likelihood (NLL), in the software it is defined thanks to the eqs. 4.6 and 3.17, during the training its derivative with respect to the weights of the network is approximated by means of finite differences.

```
# Cost for the multiple components MDN using GAMMA distribution
def gammanll_loss(y, alpha, concentration, rate):
    """ Computes the mean negative log-likelihood loss of y given
        the mixture parameters.
    """

    gm = tfd.MixtureSameFamily(
         mixture_distribution=tfd.Categorical(probs=alpha),
         components_distribution=tfd.Gamma(
              concentration=concentration,
              rate=rate))
    # Evaluate log-probability of y
    log_likelihood = gm.log_prob(tf.transpose(y)+1e-15)

    return -tf.reduce_mean(log_likelihood, axis=-1)
```

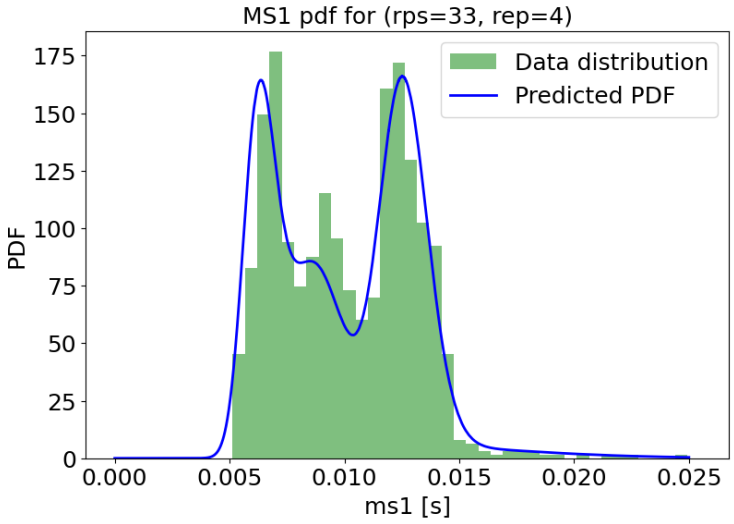
Listing 4.2: Implementation of the Gamma error function

For a first evaluation of the generalization performance of this model, figure 4.3 shows two examples of output belonging to the test set.

To summarize the performance figure 4.4 shows the Kolmogorov-Smirnov (KS) test on the entire test set: 18 out of 30 distributions shows a p-value greater than 0.05 (possible to conclude that the MDN is able to fit these distributions).



(a) $p(t_{ms1}|\lambda = 27, n_{rep} = 1)$



(b) $p(t_{ms1}|\lambda = 33, n_{rep} = 4)$

Figure 4.3: Two test pdfs

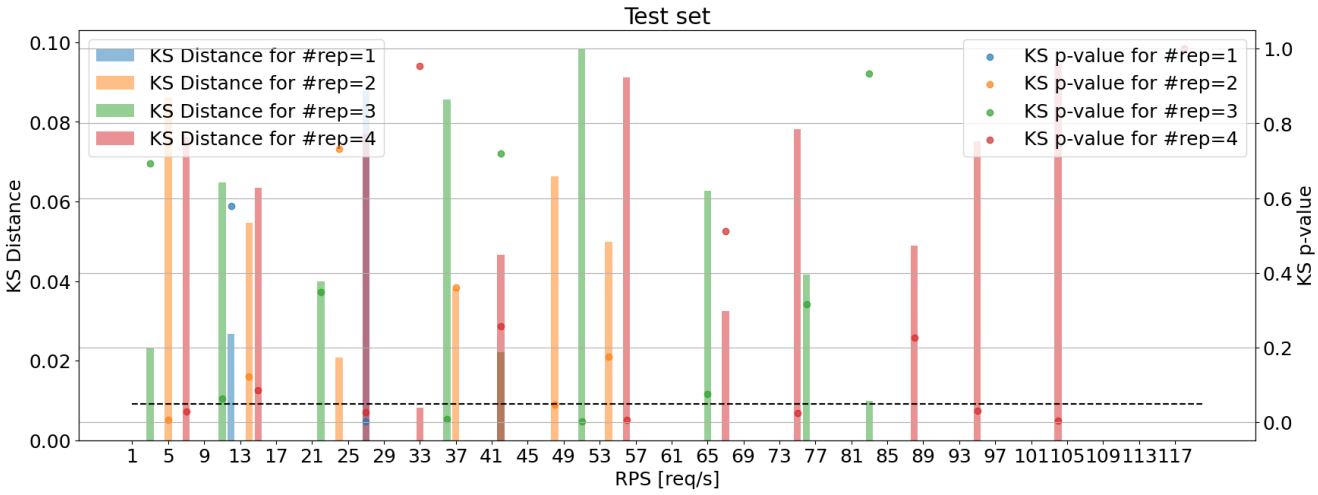


Figure 4.4: KS test for ms1, the dashed line marks the 0.05 p-value

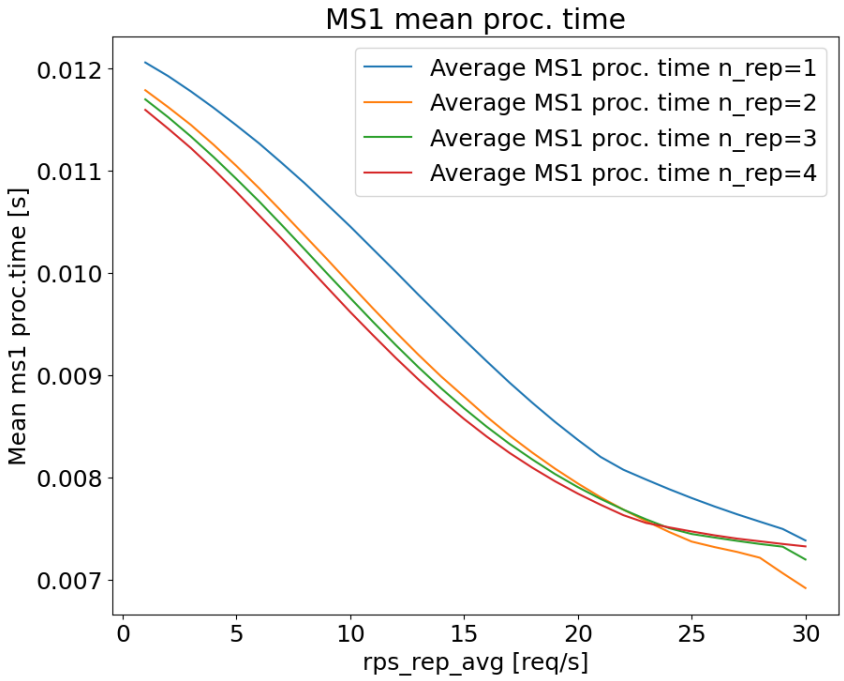
```
for rep in range(1,n_rep + 1):
    for rps in range(1,n_rep*30 + 1):
        df=data_test[(data_test['rps_rep_avg'] == rps/rep) & (
            data_test['rep'] == rep)]
        if not df.empty:
            ms1_test = df.pop('ms1')
            alpha_pred, conc_pred, rate_pred = mdn_ms1_model.
                predict(list((np.array([rps/rep,rep]),np.array
                ([8,1]))))
            gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
                Categorical(probs=alpha_pred),
                components_distribution=tfd.Gamma(concentration=
                conc_pred ,rate=rate_pred))
            testres=stats.kstest(ms1_test, cdf=mixture_cdf, args=(
                rps/rep, rep), alternative='less')
            ks_distance.append(testres.statistic)
            ks_p.append(testres.pvalue)
            rps_pl.append(rps)
```

Listing 4.3: Implementation and plot of KS test

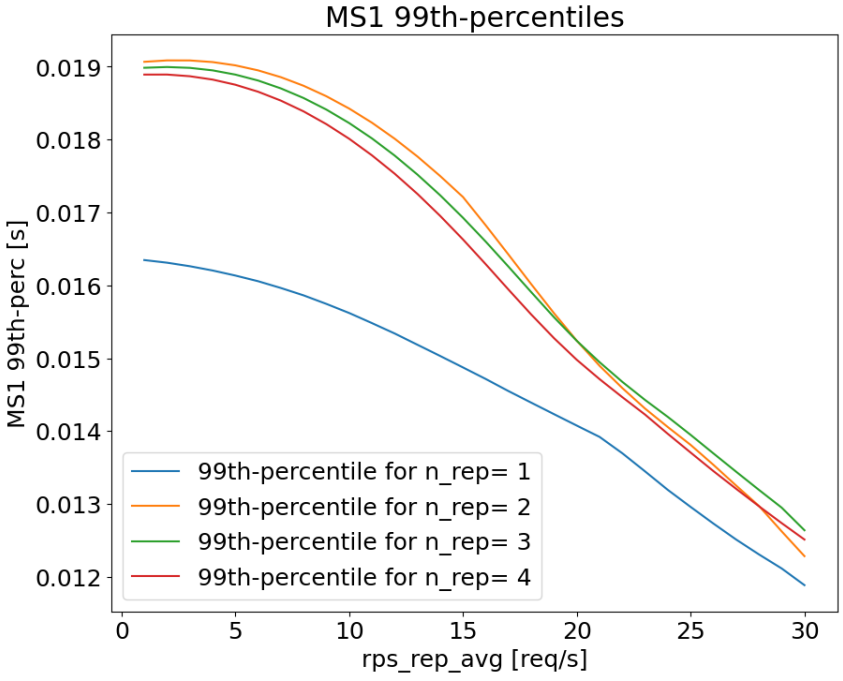
It is worth noting the distributions showing p-values less than 0.05 have maximum KS distance of 0.1 and KS test results are always pessimistic since it accounts for the maximum distance between data distributions and predictions, a more fair goodness of fit metric accounting for the could be used (e.g., a metric accounting for the average distance).

Once trained, the MDN is used to generalize the behaviour of the system and make comparisons. The figures 4.5a and 4.5b shows the comparisons of the mean and the 99-th percentile of ms1 response time in function of the average request rate per replica (defined as $rps\_rep\_avg = \frac{\lambda}{n_{rep}}$), and n_{rep}.

A first evidence is the mean response time is a decreasing function of the average request rate per replica, and this applies for all the cases. This behaviour can be indicated as a *speed-up* of the processors depending on the request rate. The second evidence is the mean response time tends to be lower when multiple replicas are available with respect the case with only one replica. The same comparison on the 99-th percentile of the response time shows that the 99-th percentile is a decreasing function of the average request rate per replica as well, however the cases with multiple replicas shows higher 99-th percentiles than the case with only one replica, this trend is opposite with respect to what seen in the mean.



(a) MS1 mean response time



(b) MS1 99-th percentile of response time

Figure 4.5: MS1 response time comparison for different cases of n_{rep}

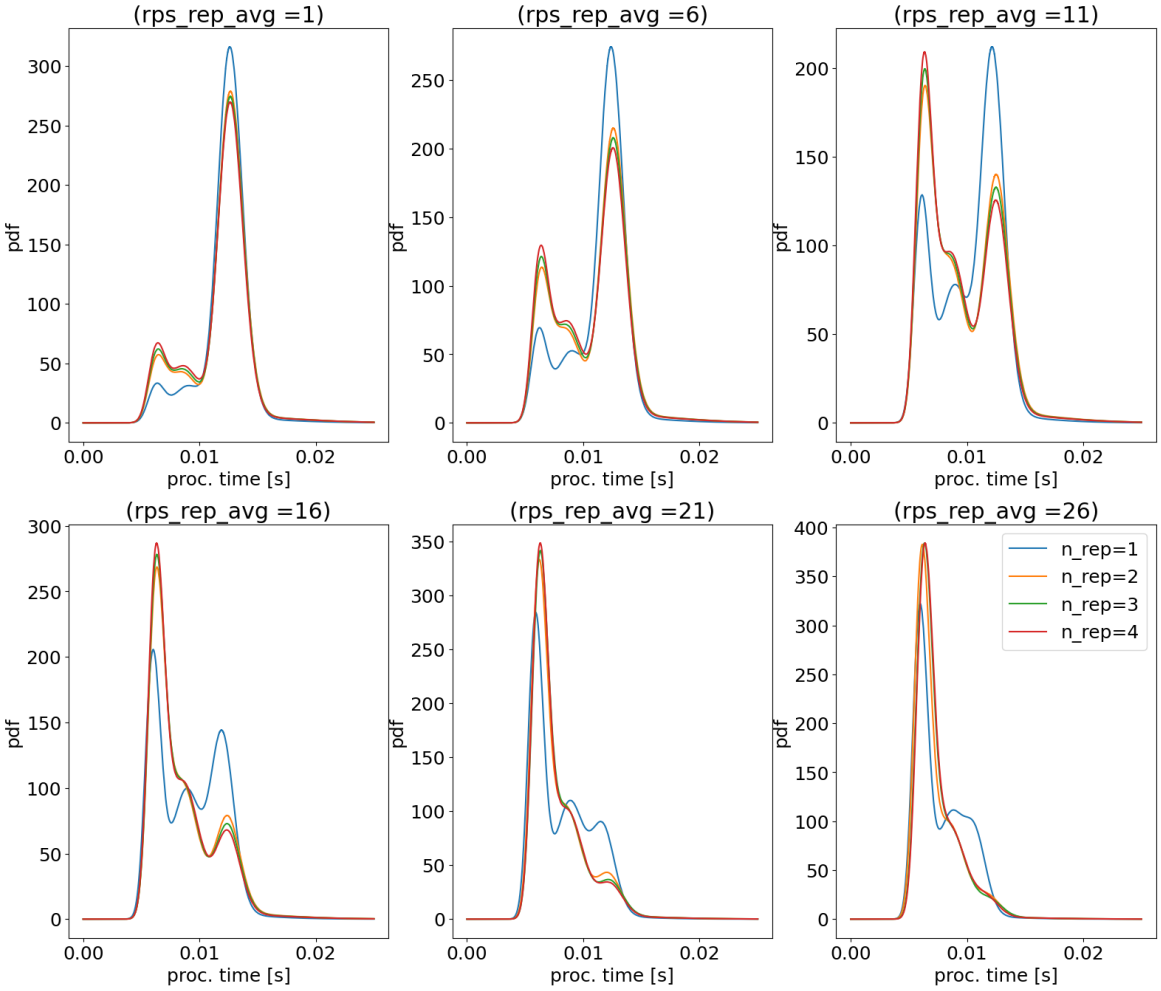


Figure 4.6: Comparison of some pdfs of ms1 response time

Focusing on the pdfs (figure 4.6) the comparison shows a bimodal tendency. In particular:

- For low values and high values of rps_rep_avg the bimodal tendency is weak, the response times are mainly distributed around the main mode.
- For intermediate values of rps_rep_avg the bimodal tendency is stronger, the ms1 processing tends to behave in two different ways (i.e., two main clusters of processing). Moreover, for cases with multiple replicas, the bimodal tendency start to arise for lower rps_rep_avg with respect the case

of one replica and is more focused on the cluster with lower processing times. and this focus is emphasized increasing rps_rep_avg.

A possible explanation for this phenomenon is an unequal and variable balancing of requests on replicas with a dynamic allocation of resources depending on the balancing: replicas fed with higher request rates belongs to the cluster with lower response times, on the other hand replicas fed with lower request rates belongs to the cluster of higher response times.

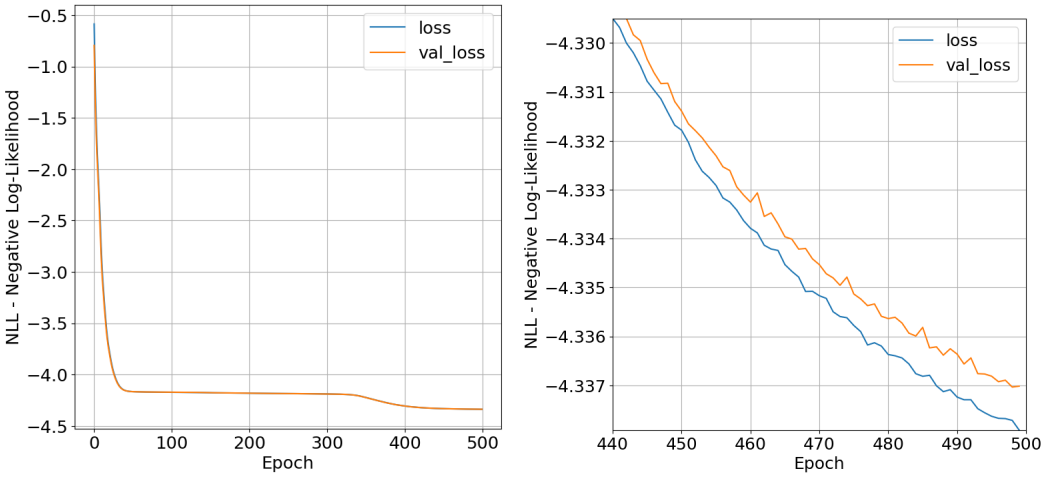
4.3.2 Model 2: characterization of microservice ms2 response time

This second model aim to characterize the microservice ms2 response time, similarly to what was done for ms1, to complete the characterization of the microservices of the system.

The objective is to approximate:

$$p(t_{ms2}|\lambda, n_{rep}) \tag{4.4}$$

The structure of the MDN and the division of the dataset for training, validation and testing is the same of model 1. For what concern the training, the MDN required 500 epochs, performed in 30 minutes with a batch size of 512 samples.



(a) Training history

(b) Zoom on the last 60 epochs

Figure 4.7: Training history of model 2. The final error is NLL=-4.337 on the validation set

The performance are summarized in figure 4.8. The KS test on the entire test set gives 29 out of 30 distributions showing a p-value greater than 0.05 (possible to conclude that the MDN is able to fit these distributions).

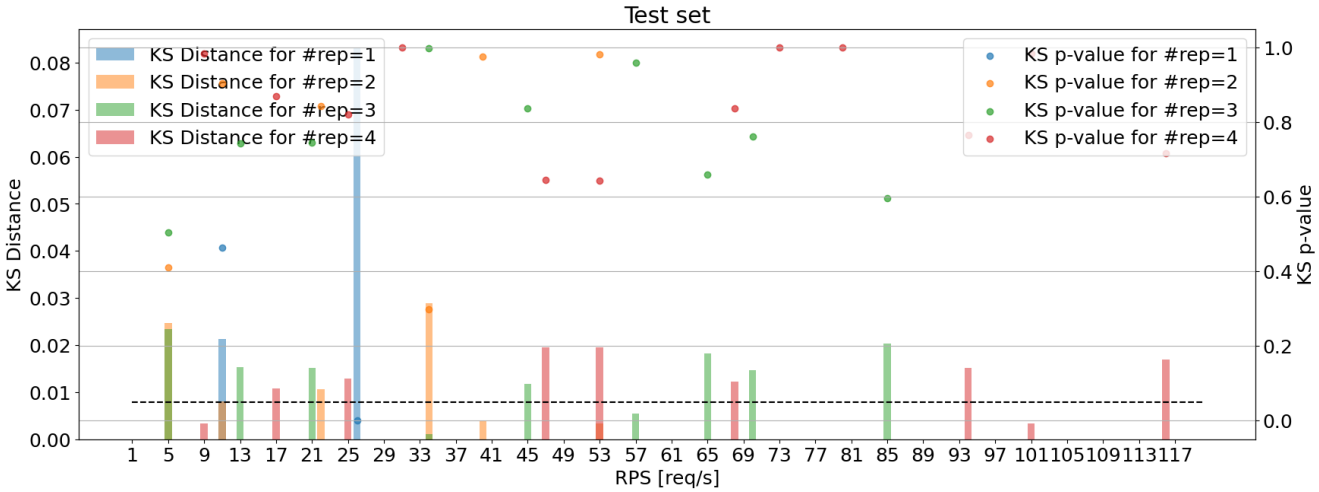


Figure 4.8: KS test for ms2, the dashed line marks the 0.05 p-value

The figure 4.9 shows the ms2 mean response time, the chart allows to do the same consideration on the speed-up made on ms1.

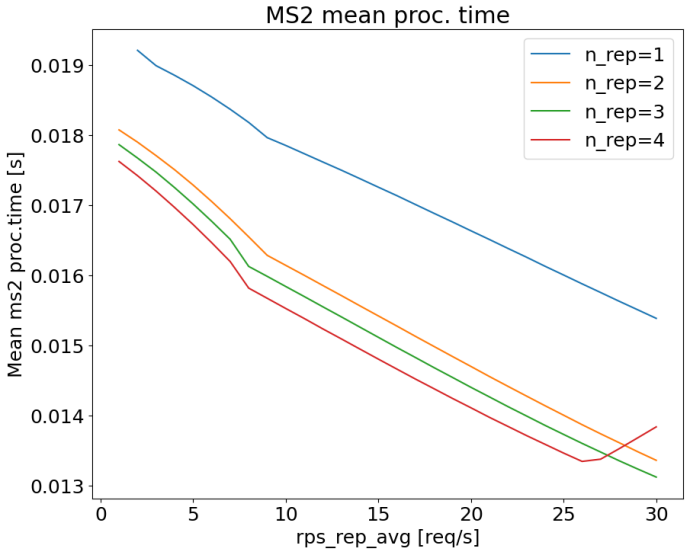


Figure 4.9: MS2 mean response time

Even for ms2 the bimodal tendency is present (figure 4.10) and the same considerations of ms1 can be done.

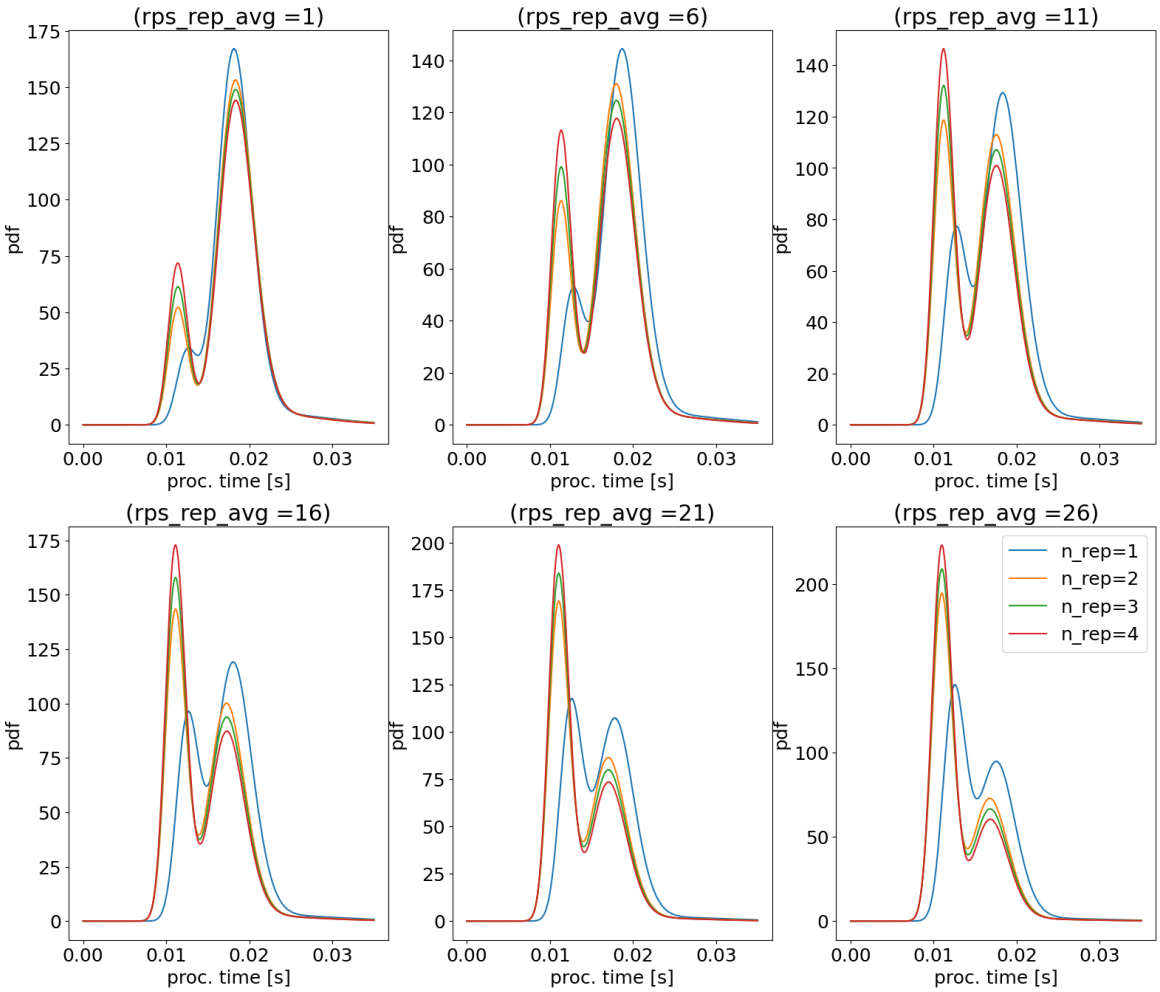


Figure 4.10: Comparison of some distributions of ms2 response time

4.3.3 Considerations about model 1 and model 2

The two models previously discussed are intended to characterize the response times of microservices ms1 and ms2. The properties of these response times have been investigate to:

- generalize the behaviour of the replicas of ms1 and ms2;
- perform analysis;
- obtain the probability density functions to make simulations.

The following conclusions may be drawn:

- The ms1 and ms2 average response times depends on the request parameters (λ, n_{rep});
- The MDN capture and generalize the dependency by means of probability density functions;
- In case the speed-up is significant, additional considerations on the trade-off between speed-up and number of replicas can be done;
- The causes of the dependency needs to be investigated, for instance additional inputs can be identified and added in the model (e.g., the ones related to the resource allocation) to reduce the uncertainty (e.g., to separate the clusters of processing).

The pdfs produced by the model are suitable to get samples of the replicas response time that can be exploited in simulations of the overall system.

A way to perform simulations of cloud-native environments is the implementation of a *Digital twin* (DT), this software component make possible the real-time optimization and diagnostic of the system developed with *Kubernetes* (k8s).

The figure 4.11 shows a work in progress about the inference procedure of the time-to-resolution of the real K8S system by its digital twin *KubeTwin*, using the pdfs obtained through the model 1 and model 2.

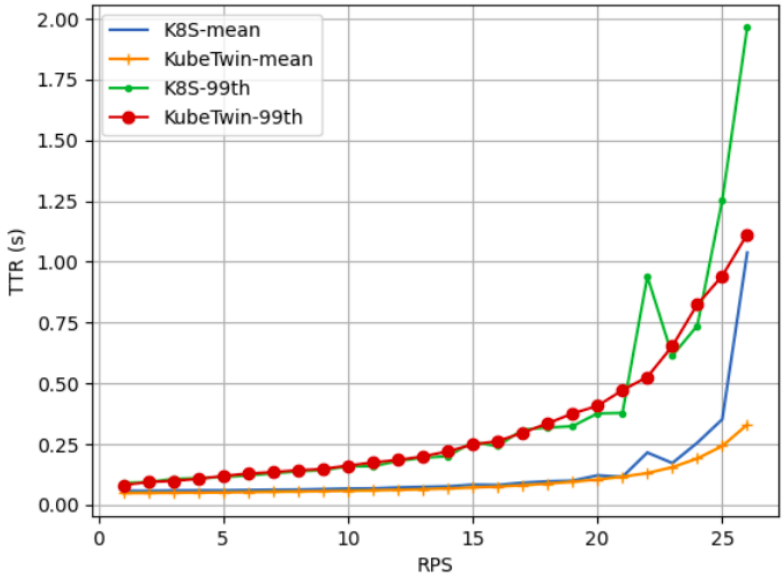


Figure 4.11: K8S system performance and predicted performance by the DT

The ability of the DT to emulate the K8S are accurate up to $\lambda = 20$ in terms of mean TTR and its 99-th percentile.

4.3.4 Model 3: characterization of time-to-resolution

This third model of MDN aims to characterize the time-to-resolution (TTR) experienced by users making requests on the system.

Mathematically, the model approximate:

$$p(ttr|\lambda, n_{rep}) \tag{4.5}$$

Where:

- $p(\bullet)$ indicates the probability density function (pdf);
- $ttr[s]$ is the time-to-resolution;
- (λ, n_{rep}) are respectively the request rate $[req/s]$ and the number of available replicas in each layer of the system.

The model is a DNN implemented with a Keras sequential model and TensorFlow probability, with the following characteristics:

- 2 input neurons: one input neuron for λ and one for n_{rep};
- 3 mixture components: the approximated pdf is the superposition of 3 Gamma's pdfs;
- 9 output neurons: the outputs represent the parameters of the components in the mixture model, every Gamma's pdf is characterized by 3 parameters (the weight in the superposition, the concentration and the rate)
- 2 hidden layer: each hidden layer has 6 neurons.

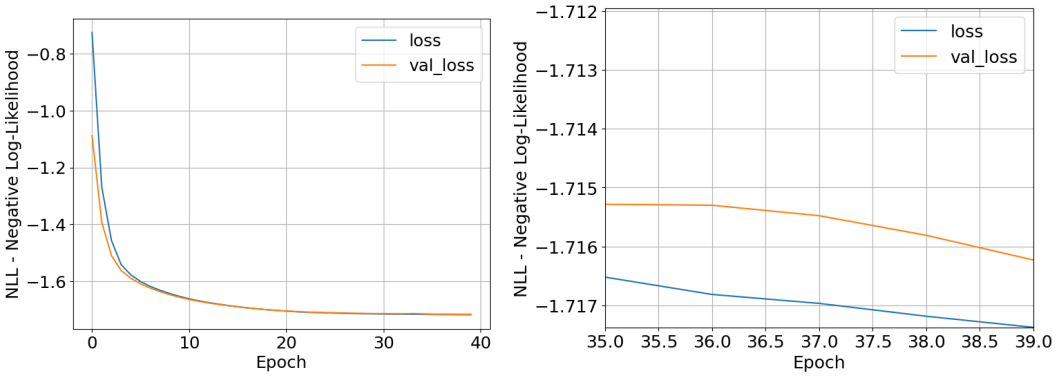
The architecture is similar to the previous models, the only difference is given by one less component in the mixture that has an impact on the number of hidden neurons too.

At the end of the training process the time-to-resolution is characterized in the following way:

$$p(ttr|\lambda, n_{rep}) = \sum_{c=1}^{3} \alpha_c(\lambda, n_{rep})\phi\big(ttr, a_c(\lambda, n_{rep}), b_c(\lambda, n_{rep})\big) \tag{4.6}$$

```
InputLayer = Input(shape=(2,))
Norm_layer = Normalization(axis = -1)
Layer_1 = Dense(6,activation="ReLU")(Norm_layer(InputLayer))
Layer_2 = Dense(6,activation="ReLU")(Layer_1)
alpha = Dense(3, activation="softmax")(Layer_2)
concentration = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1
    e-15)(Layer_2)     # The constant term 1e-15 avoids null values
    that leads to 'NaN' loss
rate = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1e-15)(
    Layer_2)
y_real = Input(shape=(1,))
lossF = gammanll_loss(y_real, alpha, concentration, rate)
mdn_ttr_model = Model(inputs=[InputLayer, y_real], outputs=[alpha,
     concentration, rate])
mdn_ttr_model.add_loss(lossF)

# Learning rate scheduler with keras.optimizers.schedules
lr_schedule =tf.keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=i_lr, decay_steps=100000,
        end_learning_rate=e_lr, power=1.0,
    cycle=False, name=None
)
adamOptimizer = optimizers.Adam(learning_rate=lr_schedule,
    clipvalue=1.0, clipnorm=1.)  # clipnorm=1 avoid gradient
    explosion (reject all gradients with norm >1)
mdn_ttr_model.summary()
mdn_ttr_model.compile(optimizer=adamOptimizer)
Norm_layer.adapt(features)
```

Listing 4.4: Model 3 implementation



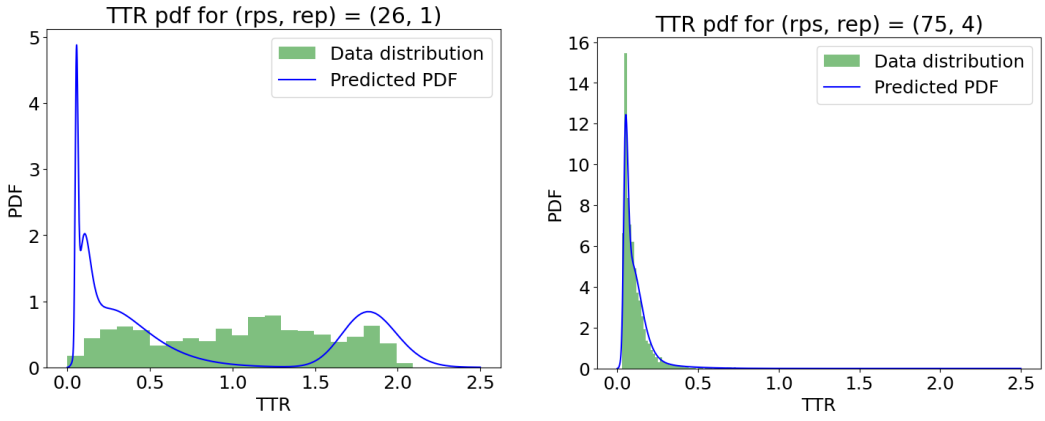
(a) Training history

(b) Zoom on the last 4 epochs

Figure 4.12: Training history of model 2. The final error is NLL=-1.7162 on the validation set

As reported in figure 4.12 the NLL converged after 40 epochs of training, requiring approx. 38 min on the previously used laptop (GPU acceleration not exploited) with a batch size of 32 samples.

A first graphical evaluation of the generalization performance of this model is shown in figure 4.13 shows two examples of output belonging to the test set. The figure 4.13a shows a bad fit of the distribution.



(a) $p(ttr|\lambda = 26, n_{rep} = 1)$

(b) $p(ttr|\lambda = 75, n_{rep} = 4)$

Figure 4.13: Two test pdfs

To summarize the performance figure 4.14 shows the Kolmogorov-Smirnov (KS) test on the entire test set: 19 out of 30 distributions shows a p-value lower than 0.05 (possible to conclude that the MDN is not able to fit these distributions). The model is characterized by an high bias. In particular, the

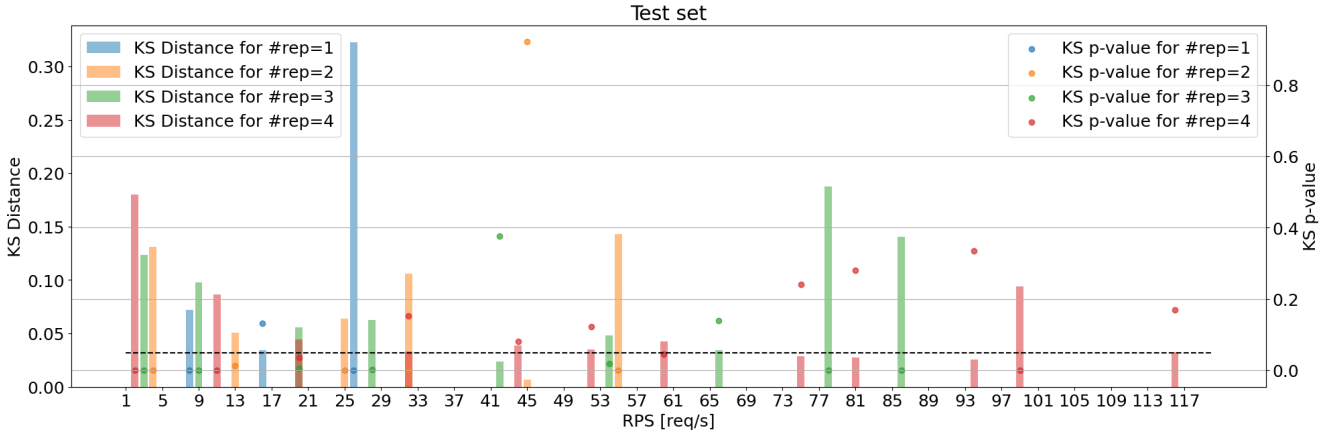


Figure 4.14: KS test for ttr, the dashed line marks the 0.05 p-value

maximum KS distance is 33% and it is related to the case $\lambda = 26, n_{rep} = 1$).

An analysis on the data highlighted the following issues:

- Non-stationarity
- Strong autocorrelation

These two properties arise for time-series related to requests parameters close to the condition $rps\_rep\_avg = \dfrac{\lambda}{n_{rep}} = 25$.

A possible explanation for this phenomenon can be that the system for this particular configuration of request parameters works in unstable conditions, meaning the service response time exceeds the inter-requests time and this bring to congestion with the queues in the system starting to grow rapidly.

To clarify, the non-stationaorty and high autocorrelation properties of an experiment with $(\lambda = 26, n_{rep} = 1)$ are reported.

The figure 4.15 shows the time-series of the time to resolution with parameters $(\lambda = 26, n_{rep} = 1)$ and the moving average, both in function of the request identifier ("rid"), a progressive number identifying the requests during

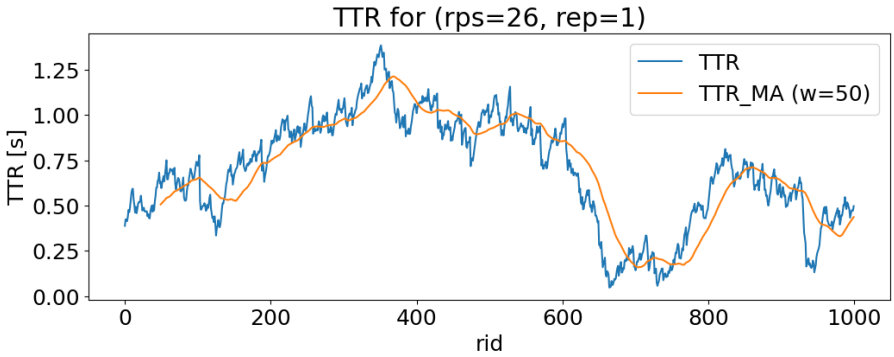


Figure 4.15: Time-series $ttr(\lambda = 26, n_{rep} = 1)$ and its moving-average computed on a window of 50 samples. The time is accounted by the request identifier.

experiments. From the figure it is possible to see the variation of the mean value of TTR during the experiment, that is an indicator of non-stationarity.

The upwards trends in the time-series are probably related to the instability condition in the system, vice versa the downward trends are related to the achievement of a stability condition due to the speed-up of the replica. The trends in the time-series turns out in a high autocorrelation (figure 4.16).

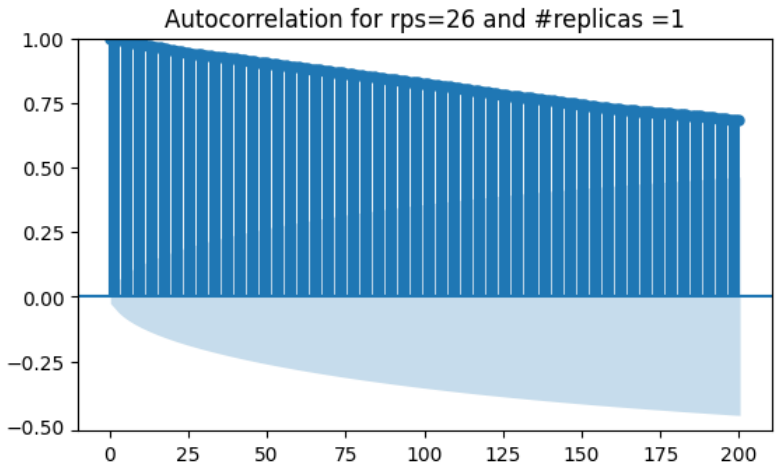


Figure 4.16: Autocorrelation function of time-series $ttr(\lambda = 26, n_{rep} = 1)$

The considered MDN model can't fit these data, since it characterize the

TTR as a random variable and its training relies on the hypothesis of independent and identically distributed samples (i.i.d.) and this hypothesis doesn't hold due to non-stationarity and high autocorrelation.

A new model able to characterize $\{TTR_i\}_{i\in\mathbb{N}}$ as a random process needs to be implemented.

4.3.5 Model 3-bis: characterization of time-to-resolution

The model 3 was extended to exploit the autocorrelation in the time-to-resolution random process. In order to implement the new model, the input space is extended with additional features representing the previous realizations of the random process; as this architecture works with previous realization, it needs to have a temporal knowledge of the system.

Mathematically, the model approximates:

$$p(ttr_i \,|\, \overline{ttr}_{i,h=5},\, \lambda,\, n_{rep}) \tag{4.7}$$

Where:

- $p(\bullet)$ indicates the probability density function (pdf);
- $ttr_i[s]$ is the ttr experienced by the i-th request;
- $\overline{ttr}_{i,h=5} = (ttr_{i-1}, \ldots, ttr_{i-5})$ is the history vector, containing the five previously experienced ttrs;
- (λ, n_{rep}) are respectively the request rate $[req/s]$ and number of available replicas for ms1.

The approximation given by the model is useful to make predictions about the ttr experienced by users given the previous five realizations. The model can be extended to larger history vector (in principle up to the maximum lags showing high autocorrelation).

The structure of the MDN requires the extension of the input layer and this have an impact on the other layers too.

The implementation is given by:

- 7 input neurons: two input neurons for λ and n_{rep} parameters, then other 5 neurons are for the history samples;

- 3 mixture components: the approximated pdf is the superposition of 3 Gamma's pdfs;
- 9 output neurons: the outputs represent the parameters of the components in the mixture model, every Gamma's pdf is characterized by 3 parameters (the weight in the superposition, the concentration and the rate);
- 2 hidden layer: each hidden layer has 8 neurons.

```
InputLayer = Input(shape=(7,))
Norm_layer = Normalization(axis=-1)
Layer_1 = Dense(8,activation="ReLU")(Norm_layer(InputLayer))
Layer_2 = Dense(8,activation="ReLU")(Layer_1)

alpha = Dense(3, activation="softmax")(Layer_2)
concentration = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1
    e-15)(Layer_2)     # The constant term 1e-15 avoids null values
    that leads to 'NaN' loss
rate = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1e-15)(
    Layer_2)
y_real = Input(shape=(1,))
lossF = gammanll_loss(y_real, alpha, concentration, rate)
mdn_ttr_model = Model(inputs=[InputLayer, y_real], outputs=[alpha,
     concentration, rate])
mdn_ttr_model.add_loss(lossF)

# Learning rate scheduler with keras.optimizers.schedules
lr_schedule =tf.keras.optimizers.schedules.PolynomialDecay(
     initial_learning_rate=i_lr, decay_steps=1e6, end_learning_rate
        =e_lr, power=1.0,
     cycle=False, name=None
)
adamOptimizer = optimizers.Adam(learning_rate=lr_schedule,
    clipvalue=1.0, clipnorm=1.)  # clipnorm=1 avoid gradient
    explosion (reject all gradients with norm >1)
mdn_ttr_model.summary()
mdn_ttr_model.compile(optimizer=adamOptimizer)
```

Listing 4.5: MDN model 3-bis implementation

The network training required 40 epochs, corresponding to a training time of 44 minutes with a batch of 32 samples.

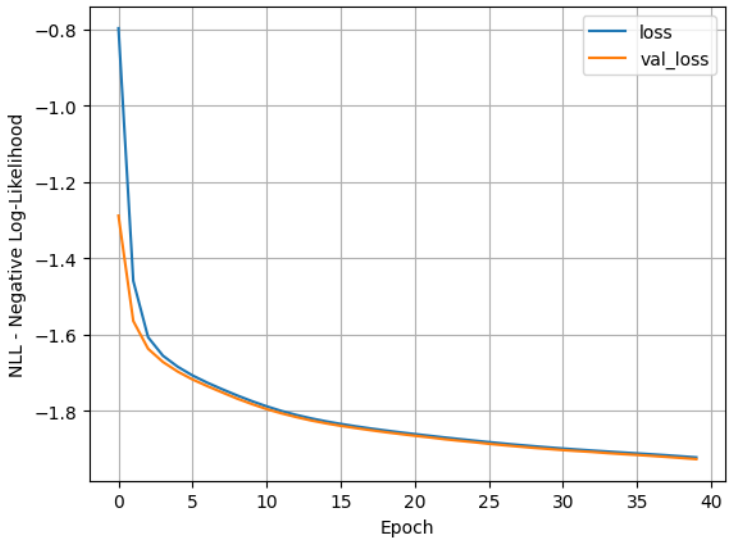


Figure 4.17: Training history of model 3-bis. The final error is NLL=-1.9266 on the validation set

```
epochs = 40
batch_size = 32
val_split=0.22
history_cache = mdn_ttr_model.fit([features, ttr], #using an input
    to pass the real values to compute the NLL
                                  verbose=1,
                                  epochs=epochs,
                                  validation_split=val_split,
                                  batch_size=batch_size)
print('Final cost: {0:.4f}'.format(history_cache.history['loss'
    ][-1]))
```

Listing 4.6: MDN model code for training

At the end of the training the model, the eq. 4.7 allows to define two functions of prediction:

- The estimator for the time-to-resolution:

$$\widehat{TTR_i} = \mathbb{E}[TTR_i \,|\, \overline{ttr}_{i,h=5},\, \lambda,\, n_{rep}] \tag{4.8}$$

- The upper-bound, corresponding to the 95-th percentile.

```
def get_mean_prediction(a):
    a= np.array(a)
    alpha_pred, conc_pred, rate_pred = mdn_ttr_model.predict(list
        ((a,a)))
    gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
        Categorical(probs=alpha_pred), components_distribution=tfd.
        Gamma(concentration=conc_pred ,rate=rate_pred))
    return gm.mean()

def quantile_mixture (p, gm):
    (lambda x: gm.cdf(x))
    return inversefunc((lambda x: gm.cdf(x)+10e-10*np.log(x)),
        y_values=p, image=[0,1] )  # logaritm makes the CDF
        strictly monotonic

def get_upper_bound(a):
    a= np.array(a)
    alpha_pred, conc_pred, rate_pred = mdn_ttr_model.predict(list
        ((a,a)))
    gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
        Categorical(probs=alpha_pred), components_distribution=tfd.
        Gamma(concentration=conc_pred ,rate=rate_pred))
    return quantile_mixture(0.95, gm)
```

Listing 4.7: Estimator and upper bound functions definitions

Differently to what done in the previous models, now the output is given by the two quantities: the estimation and the upper bound. The performance of the model can't be evaluated in the same way of the previous case because now the datasets are treated as time-series rather than i.i.d. samples (not possible to perform KS tests and plot pdf vs. data distribution).

The performance are thus evaluated through the computation of the MSE (Mean Square Error) and MAE (Mean Absolute Error) on predictions.

As an example, figure 4.18 shows the application of the model on the time-series of *ttr* for parameters ($\lambda = 26, n_{rep} = 1$)

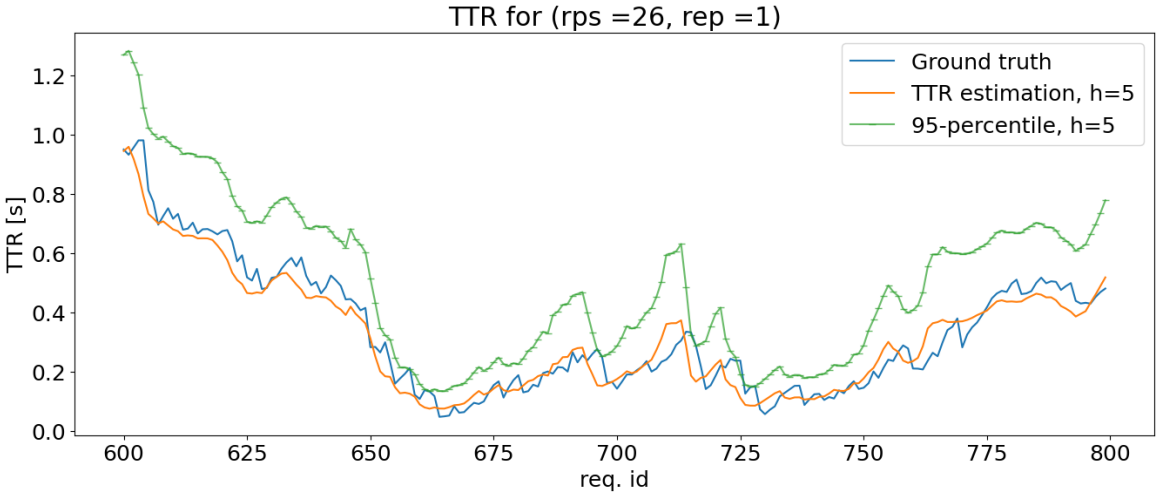


Figure 4.18: Prediction of ttr with ($\lambda = 26, n_{rep} = 1$) and upper bound.

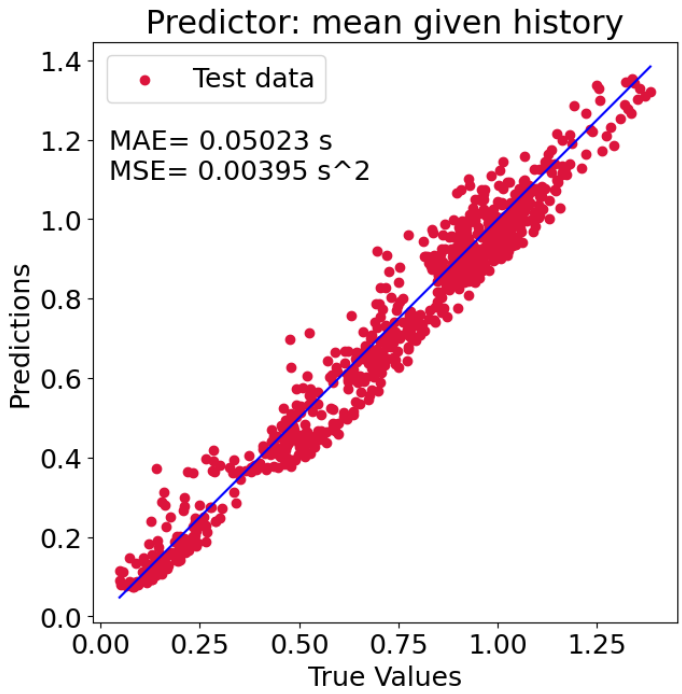


Figure 4.19: Performance (MAE and MSE) in the prediction of ttr with ($\lambda = 26, n_{rep} = 1$)

Finally, the figure 4.19 shows the performance of the model on the same example. The performance are given by $MSE = 0.00395\,s^2$ and $MAE = 0.05023\,s$. The MAE corresponds to a mean absolute percentage error of 9%.

4.3.6 Considerations about model 3 and model 3-bis

The models 3 and 3-bis are intended to characterize the time-to-resolution (ttr) of the cloud-native system. The ttr can't be characterized like the response times of the microservices due to the issues described in the previous sections; as well the model 3-bis performance can't be evaluated with KS test. The conclusions after the implementation of these two models are:

- a degradation of the performance of the model is expected if the hypothesis of i.i.d. data doesn't hold;
- the MDN can exploit the autocorrelation:
  - the ttr can be predicted based on its history if it is available;
  - the approach accounts for the time and allow the modelling of ttr as a random process;
  - the history vector can be extended;
  - the performance of the predictions are measured in terms of MAE and MSE.

4.3.7 Summary of models

This section provide a recap of the presented models.

Name	Input	Output
Model 1	(λ, n_{rep})	$p(t_{ms1}\|\lambda, n_{rep})$
Model 2	\\	$p(t_{ms2}\|\lambda, n_{rep})$
Model 3	\\	$p(ttr\|\lambda, n_{rep})$
Model 3-bis	$\overline{ttr}_{i,h=5}$, λ, n_{rep}	$p(ttr_i \mid \overline{ttr}_{i,h=5}, \lambda, n_{rep})$

Chapter 5

Conclusion and future work

In conclusion, cloud-native has become an important paradigm for large-scale application deployment, enabling efficient management of infrastructures, microservices, and distributed applications. A cloud-native deployment can be described as a multi-layered queue system, where the management with platforms such as Kubernetes is critical to ensure service availability and system scalability.

However, theoretical queueing models have limitations, which can be overcome through the use of AI models. Indeed, the use of AI can be the way to analyze the performance of microservices more precisely and generalize their behavior, as well as to use the results to do simulations in a digital twin.

In the approach proposed in this thesis, it is shown that the use of AI, applied to the system with a MDN model, can lead to adequate performance in analyzing the microservices response times and in the use of results in a digital twin. In the specific case of the digital twin, accuracy up to a rate of requests in the system not exceeding 20 [req/s] is shown.

In addition, the same kind of model to characterize the microservice response time is applied to the time-to-resolution to have a description of the service time experienced by the users; this approach results inadequate since the i.i.d. assumption does not hold. The models is then adapted to the new case, extending the input space with collection of previous experienced time-to-resolution, allowing an estimate of time-to-response and an upper bound with good performance.

However, the future direction could be to improve the models by expanding the space of input variables (e.g., by adding the variables related to resource

allocation of the replicas) and the adoption of other ML architectures, such as reinforcement learning models, or using ensemble learning techniques.

In conclusion, the use of AI in the cloud-native context offers many opportunities to improve the efficiency and scalability of distributed systems, and it is a field of research that could be widely explored in the future.

Appendix A

Some more code

A.1 Code for models 1 and 2

```
import numpy as np
import pandas as pd
import tensorflow as tf
import scipy.stats as stats
from tensorflow import keras
from tensorflow_probability import distributions as tfd
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
import keras
from keras import optimizers
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Layer, Input,
    Concatenate, Normalization, BatchNormalization
import glob

from scipy.stats import norm, kstest, gamma
from sklearn.neighbors import KernelDensity
from pynverse import inversefunc

# Data import

#Import the dataset


# Data import of replica1
path = ''
```

```
26 filenames = glob.glob(path)
27 i=1
28
29 ms1_list= []
30
31 rep=1
32
33 for f in filenames:
34     index_rps=f.find('rps') + 3
35     rps=int(float(f[index_rps:index_rps+2]))
36     print(f)
37     df = pd.read_csv(f, index_col=False, sep="\t", header = None,
           names=["rid","ms1"])
38     df.insert(1,'rps_rep_avg',rps/rep)
39     df.insert(2,'rep',rep)
40     df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
           adding moving average
41     ms1_list.append(df)
42
43 filenames = glob.glob(path)
44
45 rep=2
46
47 for f in filenames:
48     index_rps=f.find('rps') + 3
49     rps=int(float(f[index_rps:index_rps+2]))
50     print(f)
51     df = pd.read_csv(f, index_col=False, sep='\t', header = None,
           names=['rid','ms1'])
52     df.insert(1,'rps_rep_avg',rps/rep)
53     df.insert(2,'rep',rep)
54     df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
           adding moving average
55     ms1_list.append(df)
56
57 filenames = glob.glob(path)
58
59 rep=3
60
61 for f in filenames:
62     index_rps=f.find('rps') + 3
63     rps=int(float(f[index_rps:index_rps+2]))
64     print(f)
65     df = pd.read_csv(f, index_col=False, sep='\t', header = None,
           names=['rid','ms1'])
```

```
66      df.insert(1,'rps_rep_avg',rps/rep)
67      df.insert(2,'rep',rep)
68      df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
            adding moving average
69      ms1_list.append(df)
70
71  filenames = glob.glob(path)
72
73  rep=4
74
75  for f in filenames:
76      index_rps=f.find('rps') + 3
77      rps=int(float(f[index_rps:index_rps+3]))
78      print(f)
79      df = pd.read_csv(f, index_col=False, sep='\t', header = None,
            names=['rid','ms1'])
80      df.insert(1,'rps_rep_avg',rps/rep)
81      df.insert(2,'rep',rep)
82      df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
            adding moving average
83      ms1_list.append(df)
84
85  data_train_val = pd.concat(ms1_list, axis=0, ignore_index=False)
86
87
88
89  features = shuffle(data_train_val)
90
91
92  rid = features.pop('rid')
93  ms1 = features.pop('ms1')
94  MA_ms1 = features.pop('MA')
95
96
97  def plot_loss(history, zoom=False):
98    plt.plot(history.history['loss'], label='loss', )
99    plt.plot(history.history['val_loss'], label='val_loss')
100   plt.xlabel('Epoch')
101   plt.ylabel('NLL - Negative Log-Likelihood')
102   if zoom:
103     plt.xlim(350, 500)
104     plt.ylim(min(history.history['loss']), history.history['loss'
          ][350])
105   plt.legend()
106   plt.grid(True)
```

```
107
108 def gammanll_loss(y, alpha, concentration, rate):
109     """ Computes the mean negative log-likelihood loss of y given
        the mixture parameters.
110     """
111
112     gm = tfd.MixtureSameFamily(
113         mixture_distribution=tfd.Categorical(probs=alpha),
114         components_distribution=tfd.Gamma(
115             concentration=concentration,
116             rate=rate))
117
118     log_likelihood = gm.log_prob(tf.transpose(y)+1e-15) # Evaluate
        log-probability of y
119
120     return -tf.reduce_mean(log_likelihood, axis=-1)
121
122
123 InputLayer = Input(shape=(2,))  # 2 input neurons
124
125 Norm_layer = Normalization(axis = -1)
126
127 Layer_1 = Dense(7,activation="ReLU")(Norm_layer(InputLayer))    #3
128 Layer_2 = Dense(7,activation="ReLU")(Layer_1)
129 alpha = Dense(4, activation="softmax")(Layer_2)
130 concentration = Dense(4, activation=lambda x: tf.nn.elu(x) + 1 + 1
    e-15)(Layer_2)    # The constant term 1e-15 avoids null values
    that leads to 'NaN' loss
131 scale = Dense(4, activation=lambda x: tf.nn.elu(x) + 1 + 1e-15)(
    Layer_2)
132 y_real = Input(shape=(1,))
133 rate =1/scale
134 lossF = gammanll_loss(y_real, alpha, concentration, rate)  #
    gammanll_loss(y_real, alpha, mu, sigma)
135 mdn_ms1_model = Model(inputs=[InputLayer, y_real], outputs=[alpha,
     concentration, rate])
136 mdn_ms1_model.add_loss(lossF)
137
138 # Learning rate scheduler with keras.optimizers.schedules
139 lr_schedule =tf.keras.optimizers.schedules.PolynomialDecay(
140     initial_learning_rate=0.00007, end_learning_rate=0.00003,
        decay_steps=10000
141 ) #define the rate scheduler -> if we increase the number of
    layers we have to decrease the end learning rate (high
    complexity in the network)
```

```
142
143 adamOptimizer = optimizers.Adam(learning_rate=lr_schedule) # clipnorm=1 avoid gradient explosion (reject all gradients with norm >1)
144 mdn_ms1_model.summary()
145 mdn_ms1_model.compile(optimizer=adamOptimizer)
146
147 epochs = 500
148
149 history_cache = mdn_ms1_model.fit([features, ms1], #notice we are using an input to pass the real values due to the inner workings of keras
150                                   verbose=1, # write =1 if you wish to see the progress for each epoch
151                                   epochs=epochs,
152                                   validation_split=0.22,
153                                   batch_size=64)
154 print('Final cost: {0:.4f}'.format(history_cache.history['loss'][-1]))
155
156 plt.rcParams["figure.figsize"] = [8,8]
157 plot_loss(history_cache, zoom=True)
158
159
160 #Test import
161
162 filenames = glob.glob(path)
163 i=1
164
165 ms1_test= []
166
167 rep=1
168 for f in filenames:
169     index_rps=f.find('rps') + 3
170     rps=int(float(f[index_rps:index_rps+2]))
171     print(f)
172     df = pd.read_csv(f, index_col=False, sep="\t", header = None, names=["rid","ms1"])
173     df.insert(1,'rps_rep_avg',rps/rep)
174     df.insert(2,'rep',rep)
175     df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) # adding moving average
176     ms1_test.append(df)
177
178 filenames = glob.glob(path)
```

```
179
180 rep=2
181
182 for f in filenames:
183     index_rps=f.find('rps') + 3
184     rps=int(float(f[index_rps:index_rps+2]))
185     print(f)
186     df = pd.read_csv(f, index_col=False, sep='\t', header = None,
        names=['rid','ms1'])
187     df.insert(1,'rps_rep_avg',rps/rep)
188     df.insert(2,'rep',rep)
189     df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
        adding moving average
190     ms1_test.append(df)
191
192 filenames = glob.glob(path)
193
194 rep=3
195
196 for f in filenames:
197     index_rps=f.find('rps') + 3
198     rps=int(float(f[index_rps:index_rps+2]))
199     print(f)
200     df = pd.read_csv(f, index_col=False, sep='\t', header = None,
        names=['rid','ms1'])
201     df.insert(1,'rps_rep_avg',rps/rep)
202     df.insert(2,'rep',rep)
203     df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
        adding moving average
204     ms1_test.append(df)
205
206
207
208 filenames = glob.glob(path)
209
210 rep=4
211
212 for f in filenames:
213     index_rps=f.find('rps') + 3
214     rps=int(float(f[index_rps:index_rps+3]))
215     print(f)
216     df = pd.read_csv(f, index_col=False, sep='\t', header = None,
        names=['rid','ms1'])
217     df.insert(1,'rps_rep_avg',rps/rep)
218     df.insert(2,'rep',rep)
```

```
219     df.insert(3, 'MA', df.get('ms1').rolling(window=200).mean()) #
            adding moving average
220     ms1_test.append(df)
221
222 data_test = pd.concat(ms1_test, axis=0, ignore_index=False)
223
224
225 def mixture_pdf(x,rps_rep_avg, rep):
226
227     """ pdf function of mixture is the weighted superimposition of
            the pdfs """
228
229     alpha_pred, conc_pred, rate_pred = mdn_ms1_model.predict(list
            ((np.array([rps_rep_avg,rep]),np.array([1,1])))) # The
            second array is dummy
230     pdf = 0
231     for i in range(0, alpha_pred.shape[1]):
232         pdf = pdf + alpha_pred[0][i]*stats.gamma.pdf(x, a=
                conc_pred[0][i], scale=1/rate_pred[0][i])
233     return pdf
234
235 def mixture_cdf(x,rps_rep_avg, rep):
236
237     """ cdf function of mixture is the weighted superimposition of
            the cdfs """
238
239     alpha_pred, conc_pred, rate_pred = mdn_ms1_model.predict(list
            ((np.array([rps_rep_avg,rep]),np.array([1,1])))) # The
            second array is dummy
240     cdf = 0
241     for i in range(0, alpha_pred.shape[1]):
242         cdf = cdf + alpha_pred[0][i]*stats.gamma.cdf(x, a=
                conc_pred[0][i], scale=1/rate_pred[0][i])
243     return cdf
244
245 # Let's try to plot the predicted distribution for an arbitrary
        rps and rep and compare it with the distribution given in the
        test set
246
247 rps =27 # The rps
248 rep=  1  # The rep
249
250 plt.rcParams["figure.figsize"] = [9,6]
251
```

```
252 df=data_test[(data_test['rps_rep_avg'] == rps/rep) & (data_test['
        rep'] == rep)]
253
254 ms1_test = df.pop('ms1')
255 ms1_min = ms1_test.min()
256 ms1_max = ms1_test.max()
257
258 bw = ms1_test.std() *0.182 # Compute sub-optimal bandwidth for the
        histogram
259 bins = np.arange(ms1_min, ms1_max,bw)
260
261
262 # Plot PDF.
263 x_p = np.linspace(0., 0.025, int(1e5), dtype=np.float32)
264 X_r=x_p.reshape(-1, 1)
265
266 plt.hist(ms1_test.values, bins=bins, fc='g', alpha = 0.5, density=
        True, label = 'Data distribution')
267 alpha_pred, conc_pred, rate_pred = mdn_ms1_model.predict(list((np.
        array([rps/rep,rep]),np.array([1,1])))) # The second array is
        dummy
268 gm = tfd.MixtureSameFamily(mixture_distribution=tfd.Categorical(
        probs=alpha_pred), components_distribution=tfd.Gamma(
        concentration=conc_pred ,rate=rate_pred))
269 #plt.plot(x_p, mixture_pdf(x_p, rps/rep, rep), label = "Predicted
        PDF", color = 'b')
270 plt.plot(x_p, gm.prob(x_p), label = "Predicted PDF", color = 'b',
        linewidth=2)
271 #for i in range(0, alpha_pred.shape[1]):
272     #gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
            Categorical(probs=[alpha_pred[0][i]]),
            components_distribution=tfd.Gamma(concentration=[conc_pred
            [0][i]] ,rate=[rate_pred[0][i]]))
273     #plt.plot(x_p, gm.prob(x_p), label = "Predicted PDF" + str(i)
            + "component" + "prob = " + str([alpha_pred[0][i]]))
274 plt.title('MS1 pdf for (rps=' + str(rps) + ', rep=' + str(rep) +
        ')', fontsize=18)
275 plt.xlabel("ms1 [s]")
276 plt.ylabel("PDF")
277
278 plt.legend()
279
280 # Perform KS-test
281 plt.rcParams["figure.figsize"] = [20,7]
282
```

```
283 n_rep = 4
284
285 rps_pl = []
286 ks_distance = []
287 ks_p =[]
288 fig, ax1 = plt.subplots()
289
290 ax2 = ax1.twinx()
291
292 for rep in range(1,n_rep + 1):
293     for rps in range(1,n_rep*30 + 1):
294         df=data_test[(data_test['rps_rep_avg'] == rps/rep) & (
                data_test['rep'] == rep)]
295         if not df.empty:
296             ms1_test = df.pop('ms1')
297             alpha_pred, conc_pred, rate_pred = mdn_ms1_model.
                    predict(list((np.array([rps/rep,rep]),np.array
                    ([8,1]))))
298             gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
                    Categorical(probs=alpha_pred),
                    components_distribution=tfd.Gamma(concentration=
                    conc_pred ,rate=rate_pred))
299             testres=stats.kstest(ms1_test, cdf=mixture_cdf, args=(
                    rps/rep, rep), alternative='less')
300             ks_distance.append(testres.statistic)
301             ks_p.append(testres.pvalue)
302             rps_pl.append(rps)
303     ax1.bar(rps_pl, ks_distance, label = "KS Distance for #rep=" +
            str(rep), alpha = 0.5)
304     ax2.scatter(rps_pl, ks_p, label = "KS p-value for #rep=" + str
            (rep), alpha = 0.7)
305     rps_pl.clear()
306     ks_distance.clear()
307     ks_p.clear()
308 ax2.plot(np.arange(1, 121), np.full(120, 0.05), 'k--')
309 plt.xticks(np.arange(1, 120, 4.0))
310 plt.grid(visible=True)
311 plt.title('Test set')
312 ax1.set_xlabel("RPS [req/s]")
313 ax1.set_ylabel("KS Distance")
314 ax2.set_ylabel("KS p-value")
315 ax1.legend(loc='upper left')
316 ax2.legend(loc='upper right')
317 plt.show()
```

Listing A.1: Code for ms1 characterization

A.2 Code for model 3

```
import numpy as np
import pandas as pd
import tensorflow as tf
import scipy.stats as stats
#from tensorflow import keras. layers
from tensorflow import keras
from tensorflow_probability import distributions as tfd
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from keras import optimizers
from keras.models import Sequential, Model
from keras.layers import Dense, Activation, Layer, Input,
    Concatenate, Normalization, BatchNormalization

import glob

from scipy.stats import norm, kstest, gamma
from sklearn.neighbors import KernelDensity
from pynverse import inversefunc

# Train data import

#Import the dataset


datalist= []

i=1
# Data import of replica1

path=''

filenames = glob.glob(path)

for f in filenames:
    print(f)
    index_rps=f.find('rps') + 3
```

```
37      df = pd.read_csv(f, index_col=0)
38      df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
            Add in the firs column the corresponding rps
39      print(int(float(f[index_rps:index_rps+3])))
40      df.insert(2,'rep',1)    # Add in the second column the
            corresponding number of replicas
41      datalist.append(df)     # Append the dataframe containing the
            values for each rps in the datalist
42      i=i+1
43
44
45  # Data import of replica2
46  filenames = glob.glob(path)
47
48  i=1
49  for f in filenames:
50      print(f)
51      index_rps=f.find('rps') + 3
52      df = pd.read_csv(f, index_col=0)
53      df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
            Add in the firs column the corresponding rps
54      print(int(float(f[index_rps:index_rps+3])))
55      df.insert(2,'rep',2)    # Add in the second column the
            corresponding number of replicas
56      datalist.append(df)     # Append the dataframe containing the
            values for each rps in the datalist
57      i=i+1
58
59  # Data import of replica3
60  filenames = glob.glob(path)
61
62  i=1
63  for f in filenames:
64      print(f)
65      index_rps=f.find('rps') + 3
66      df = pd.read_csv(f, index_col=0)
67      df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
            Add in the firs column the corresponding rps
68      print(int(float(f[index_rps:index_rps+3])))
69      df.insert(2,'rep',3)    # Add in the second column the
            corresponding number of replicas
70      datalist.append(df)     # Append the dataframe containing the
            values for each rps in the datalist
71      i=i+1
72
```

```
73 # Data import of replica4
74 filenames = glob.glob(path)
75
76 i=1
77 for f in filenames:
78     print(f)
79     index_rps=f.find('rps') + 3
80     df = pd.read_csv(f, index_col=0)
81     df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
           Add in the firs column the corresponding rps
82     print(int(float(f[index_rps:index_rps+3])))
83     df.insert(2,'rep',4)    # Add in the second column the
           corresponding number of replicas
84     datalist.append(df)     # Append the dataframe containing the
           values for each rps in the datalist
85     i=i+1
86
87
88 data_train_val= pd.concat(datalist, axis=0, ignore_index=True)  #
       Concatenate the datalist in a dataframe for semplicity
89
90 #data_train_val.head()
91
92 # Make a copy
93 train_val_copy = data_train_val.copy()  # make a copy
94
95 features = shuffle(data_train_val)
96
97
98 rid = features.pop('rid')
99 ttr = features.pop('ttr')
100
101
102 def plot_loss(history, zoom=False):
103   plt.plot(history.history['loss'], label='loss', )
104   plt.plot(history.history['val_loss'], label='val_loss')
105   plt.xlabel('Epoch')
106   plt.ylabel('NLL - Negative Log-Likelihood')
107   if zoom:
108     plt.xlim(35, 39)
109     plt.ylim(min(history.history['loss']), history.history['loss'
          ][35])
110   plt.legend()
111   plt.grid(True)
112
```

```
113
114 # GNLL loss for multiple components MDN using the Gaussian
        distributions
115 def gnll_loss(y, alpha, mu, sigma):
116     """ Computes the mean negative log-likelihood loss of y given
            the mixture parameters.
117     """
118
119     gm = tfd.MixtureSameFamily(
120         mixture_distribution=tfd.Categorical(probs=alpha),
121         components_distribution=tfd.Normal(
122             loc=mu,
123             scale=sigma))
124
125     log_likelihood = gm.log_prob(tf.transpose(y)) # Evaluate log-
            probability of y
126
127     return -tf.reduce_mean(log_likelihood, axis=-1)
128
129
130
131
132
133 InputLayer = Input(shape=(2,))
134 Norm_layer = Normalization(axis = -1)
135 Layer_1 = Dense(6,activation="ReLU")(Norm_layer(InputLayer))
136 Layer_2 = Dense(6,activation="ReLU")(Layer_1)
137
138 alpha = Dense(3, activation="softmax")(Layer_2)
139 concentration = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1
        e-15)(Layer_2)    # The constant term 1e-15 avoids null values
        that leads to 'NaN' loss
140 rate = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1e-15)(
        Layer_2)
141 y_real = Input(shape=(1,))
142 lossF = gammanll_loss(y_real, alpha, concentration, rate)
143 mdn_ttr_model = Model(inputs=[InputLayer, y_real], outputs=[alpha,
        concentration, rate])
144 mdn_ttr_model.add_loss(lossF)
145
146 # Learning rate scheduler with keras.optimizers.schedules
147 lr_schedule =tf.keras.optimizers.schedules.PolynomialDecay(
148     initial_learning_rate=i_lr, decay_steps=100000,
            end_learning_rate=e_lr, power=1.0,
149     cycle=False, name=None
```

```
)

adamOptimizer = optimizers.Adam(learning_rate=lr_schedule,
    clipvalue=1.0, clipnorm=1.) # clipnorm=1 avoid gradient
    explosion (reject all gradients with norm >1)
mdn_ttr_model.summary()
mdn_ttr_model.compile(optimizer=adamOptimizer)
Norm_layer.adapt(features)


epochs = 40 #20

history_cache = mdn_ttr_model.fit([features, ttr], #notice we are
    using an input to pass the real values due to the inner
    workings of keras
                            verbose=1, # write =1 if you wish to see
                                the progress for each epoch
                            epochs=epochs,
                            validation_split=0.22,
                            batch_size=32)
print('Final cost: {0:.4f}'.format(history_cache.history['loss'
    ][-1]))
```

Listing A.2: Code for ttr characterization

A.3 Code for model 3-bis

```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import tensorflow as tf
import scipy.stats as stats
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow_probability import distributions as tfd
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn import metrics
from keras import optimizers
from keras.models import Sequential, Model
```

```
from keras.layers import Dense, Activation, Layer, Input,
    Concatenate, Normalization
import keras
import glob

from scipy.stats import norm, kstest, gamma
from pynverse import inversefunc

plt.rcParams.update({'font.size': 18})

# Data import

#Import the dataset

path=''
datalist= []

i=1
h=5
# Data import of replica1

filenames = glob.glob(path)
for f in filenames:
    print(f)
    index_rps=f.find('rps') + 3
    df = pd.read_csv(f, index_col=None)
    df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
        Add in the firs column the corresponding rps
    print(int(float(f[index_rps:index_rps+3])))
    df.insert(2,'rep',1)    # Add in the second column the
        corresponding number of replicas
    df1 = df.copy()
    for j in range(1,h+1):
        df.insert(3 + j, 'ttr_i-' + str(j) ,df1.get('ttr').shift(j
            )) #Add in the columns from 4 to 8 the TTRs
            experienced in the previous h requests
    df = df.drop(index=range(0,h))
    datalist.append(df)     # Append the dataframe containing the
        values for each rps in the datalist
    i=i+1


# Data import of replica2
filenames = glob.glob(path)

```

```
55 i=1
56 for f in filenames:
57     print(f)
58     index_rps=f.find('rps') + 3
59     df = pd.read_csv(f, index_col=None)
60     df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
          Add in the firs column the corresponding rps
61     print(int(float(f[index_rps:index_rps+3])))
62     df.insert(2,'rep',2)    # Add in the second column the
          corresponding number of replicas
63     df1 = df.copy()
64     for j in range(1,h+1):
65         df.insert(3 + j, 'ttr_i-' + str(j) ,df1.get('ttr').shift(j
              )) #Add in the columns from 4 to 8 the TTRs
              experienced in the previous h requests
66     df = df.drop(index=range(0,h))
67     datalist.append(df)     # Append the dataframe containing the
          values for each rps in the datalist
68     i=i+1
69
70 # Data import of replica3
71 filenames = glob.glob(path)
72
73 i=1
74 for f in filenames:
75     print(f)
76     index_rps=f.find('rps') + 3
77     df = pd.read_csv(f, index_col=None)
78     df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
          Add in the firs column the corresponding rps
79     print(int(float(f[index_rps:index_rps+3])))
80     df.insert(2,'rep',3)    # Add in the second column the
          corresponding number of replicas
81     df1 = df.copy()
82     for j in range(1,h+1):
83         df.insert(3 + j, 'ttr_i-' + str(j) ,df1.get('ttr').shift(j
              )) #Add in the columns from 4 to 8 the TTRs
              experienced in the previous h requests
84     df = df.drop(index=range(0,h))
85     datalist.append(df)     # Append the dataframe containing the
          values for each rps in the datalist
86     i=i+1
87
88 # Data import of replica4
89 filenames = glob.glob(path)
```

```
90
91  i=1
92  for f in filenames:
93      print(f)
94      index_rps=f.find('rps') + 3
95      df = pd.read_csv(f, index_col=None)
96      df.insert(1,'rps',int(float(f[index_rps:index_rps+3])))    #
            Add in the firs column the corresponding rps
97      print(int(float(f[index_rps:index_rps+3])))
98      df.insert(2,'rep',4)    # Add in the second column the
            corresponding number of replicas
99      df1 = df.copy()
100     for j in range(1,h+1):
101         df.insert(3 + j, 'ttr_i-' + str(j) ,df1.get('ttr').shift(j
                )) #Add in the columns from 4 to 8 the TTRs
                experienced in the previous h requests
102     df = df.drop(index=range(0,h))
103     datalist.append(df)     # Append the dataframe containing the
            values for each rps in the datalist
104     i=i+1
105
106
107 data_train_val= pd.concat(datalist, axis=0, ignore_index=True) #
        Concatenate the datalist in a dataframe for semplicity
108
109 #data_train_val.head()
110
111 # Make some copies
112 train_val_copy = data_train_val.copy()  # make a copy
113
114 features = shuffle(data_train_val)
115
116 rid = features.pop('rid')
117 ttr = features.pop('ttr')
118 st = features.pop('st')
119
120
121 # Cost for the multiple components MDN using GAMMA distribution
122 def gammanll_loss(y, alpha, concentration, rate):
123     """ Computes the mean negative log-likelihood loss of y given
            the mixture parameters.
124     """
125
126     gm = tfd.MixtureSameFamily(
127         mixture_distribution=tfd.Categorical(probs=alpha),
```

```
128             components_distribution=tfd.Gamma(
129                 concentration=concentration,
130                 rate=rate))
131 
132         log_likelihood = gm.log_prob(tf.transpose(y)+1e-15) # Evaluate
                log-probability of y np.clip(tf.transpose(y), 1e-8, 100.)
133 
134         return -tf.reduce_mean(log_likelihood, axis=-1)
135 
136 def plot_loss(history, zoom=False):
137   plt.plot(history.history['loss'], label='loss')
138   plt.plot(history.history['val_loss'], label='val_loss')
139   plt.xlabel('Epoch')
140   plt.ylabel('NLL - Negative Log-Likelihood')
141   if zoom:
142     plt.ylim(min(history.history['loss']), min(history.history['
          loss']) + 100)
143   plt.legend()
144   plt.grid(True)
145 
146 
147 any_nan = np.any(np.isnan(ttr))|np.any(np.isnan(features.get('rps'
      )))|np.any(np.isnan(features.get('rep')))
148 for j in range(1,h+1):
149     any_nan = any_nan | np.any(np.isnan(features.get('ttr_i-' +
          str(j))))
150 
151 print('Presence of NaN: ' + str(any_nan))
152 
153 
154 InputLayer = Input(shape=(7,))  # 7input neurons
155 
156 
157 
158 Norm_layer = Normalization(axis=-1)
159 
160 Layer_1 = Dense(8,activation="ReLU")(Norm_layer(InputLayer))    #2
161 Layer_2 = Dense(8,activation="ReLU")(Layer_1)       #2)
162 
163 alpha = Dense(3, activation="softmax")(Layer_2)
164 concentration = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1
      e-15)(Layer_2)    # The constant term 1e-15 avoids null values
      that leads to 'NaN' loss
165 rate = Dense(3, activation=lambda x: tf.nn.elu(x) + 1 + 1e-15)(
      Layer_2)
```

```
166 y_real = Input(shape=(1,))
167 lossF = gammanll_loss(y_real, alpha, concentration, rate) #
        gammanll_loss(y_real, alpha, mu, sigma)
168 mdn_ttr_model = Model(inputs=[InputLayer, y_real], outputs=[alpha,
        concentration, rate])
169 mdn_ttr_model.add_loss(lossF)
170
171 # Learning rate scheduler with keras.optimizers.schedules
172 lr_schedule =tf.keras.optimizers.schedules.PolynomialDecay(
173     initial_learning_rate=0.0003, decay_steps=20,
            end_learning_rate=0.00003, power=1.0,
174     cycle=False, name=None
175 ) #define the rate scheduler -> if we increase the number of
        layers we have to decrease the end learning rate (high
        complexity in the network)
176
177 adamOptimizer = optimizers.Adam(learning_rate=lr_schedule,
        clipvalue=1.0, clipnorm=1.) # clipnorm=1 avoid gradient
        explosion (reject all gradients with norm >1)
178 mdn_ttr_model.summary()
179 mdn_ttr_model.compile(optimizer=adamOptimizer)
180
181 Norm_layer.adapt(features) # Perform normalization of input
        features to improve the convergence of the MDN
182
183 epochs = 40 #epochs = 30
184 batch_size = 32
185 history_cache = mdn_ttr_model.fit([features, ttr], #using an input
        to pass the real values to compute the NLL
186                                   verbose=1,
187                                   epochs=epochs,
188                                   validation_split=0.22,
189                                   batch_size=batch_size)
190 print('Final cost: {0:.4f}'.format(history_cache.history['loss'
        ][-1]))
191
192 def quantile_mixture (p, gm):
193     return inversefunc((lambda x: gm.cdf(x)+10e-7*np.log(x)),
            y_values=p, image=[0,1] ) # logaritm makes the CDF
            strictly monotonic
194
195 def get_mean_prediction(a):
196     a= np.array(a)
197     alpha_pred, conc_pred, rate_pred = mdn_ttr_model.predict(list
            ((a,a)))
```

```
198     gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
            Categorical(probs=alpha_pred), components_distribution=tfd.
            Gamma(concentration=conc_pred ,rate=rate_pred))
199     return gm.mean()
200
201 def get_upper_bound(a):
202     a= np.array(a)
203     alpha_pred, conc_pred, rate_pred = mdn_ttr_model.predict(list
            ((a,a)))
204     gm = tfd.MixtureSameFamily(mixture_distribution=tfd.
            Categorical(probs=alpha_pred), components_distribution=tfd.
            Gamma(concentration=conc_pred ,rate=rate_pred))
205     return quantile_mixture(0.95, gm)
206
207
208 plt.rcParams["figure.figsize"] = [16,6]
209 rps = 26
210 rep= 1
211
212 filename=''
213
214 df_plt = pd.read_csv(filename, index_col=None)
215 st_test =df_plt.pop('st')
216 ttr_test = df_plt.pop('ttr')
217
218 plt.plot(np.arange(600,800), ttr_test[600:800], label = 'Ground
        truth')
219 plt.plot(np.arange(600,800), ttr_pred_mean[600:800], label = 'TTR
        estimation, h=5')
220 plt.plot(np.arange(600,800), ttr_upper_bounds[600:800], label = '
        95-percentile, h=5', marker='_', alpha=0.7)
221
222 plt.xlabel("req. id")
223 plt.ylabel('TTR [s]')
224 plt.title('TTR for (rps =' + str(rps) +', rep ='+ str(rep) + ')')
225 plt.legend()
226
227
228 plt.figure(figsize=(7,7))
229 plt.scatter(ttr_test[5:800], ttr_pred_mean[0:795], c='crimson',
        label="Test data")
230
231
232 p1 = max(max(ttr_pred_mean[0:795]), max(ttr_test[5:800]))
233 p2 = min(min(ttr_pred_mean[0:795]), min(ttr_test[5:800]))
```

```
234 plt.plot([p1, p2], [p1, p2], 'b-')
235 plt.xlabel('True Values', fontsize=18)
236 plt.ylabel('Predictions', fontsize=18)
237 plt.axis('equal')
238 t=("MAE= " + str(round(mae_mean, 5)) + " s" "\nMSE= " + str(round(
        mse_mean, 5)) + " s^2")
239 plt.text(0.02, 1.1, s=t)
240 plt.legend()
241 plt.title("Predictor: mean given history")
242 plt.show()
```

Listing A.3: Code for ttr with history characterization

Bibliography

[1] José Santos, Tim Wauters, Bruno Volckaert, and Filip De Turck. "Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications". In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. 2019, pp. 351–359. DOI: 10.1109/NETSOFT.2019.8806671.

[2] Walter Cerroni, Luca Foschini, Genady Ya Grabarnik, Filippo Poltronieri, Larisa Shwartz, Cesare Stefanelli, and Mauro Tortonesi. "BDMaaS+: Business-Driven and Simulation-Based Optimization of IT Services in the Hybrid Cloud". In: *IEEE Transactions on Network and Service Management* 19.1 (2022), pp. 322–337. DOI: 10.1109/TNSM.2021.3110139.

[3] Xiaoyu Su, Zehan Jia, Zhenyu Zhou, Zhong Gan, Xiaoyan Wang, and Shahid Mumtaz. "Digital Twin-Empowered Communication Network Resource Management for Low-Carbon Smart Park". In: *ICC 2022 - IEEE International Conference on Communications*. 2022, pp. 2942–2947. DOI: 10.1109/ICC45855.2022.9838789.

[4] Robvet. *Cloud native explained*. URL: https://learn.microsoft.com/it-it/dotnet/architecture/cloud-native/definition.

[5] *Containerization explained*. URL: https://www.ibm.com/topics/containerization.

[6] *Containers vs VMS*. URL: https://www.redhat.com/en/topics/containers/containers-vs-vms.

[7] *Kubernetes Components*. 2022. URL: https://kubernetes.io/docs/concepts/overview/components/.

[8] Margery Meyer. *Innovative trends in 6G ecosystems*. 2022. URL: https://ieeeaccess.ieee.org/open-special-sections/innovative-trends-in-6g-ecosystems/.

[9] *One6G group white paper - 6G technology overview*. URL: https://one6g.org/download/2001/.

[10] Christopher M. Bishop. "Neural networks and their applications". In: *Review of Scientific Instruments* 65 (1994), pp. 1803–1832.

[11] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

[12] Christopher M. Bishop. *Mixture density networks*. English. WorkingPaper. Aston University, 1994.

[13] Frank J. Massey. "The Kolmogorov-Smirnov Test for Goodness of Fit". In: *Journal of the American Statistical Association* 46.253 (1951), pp. 68–78. ISSN: 01621459.

[14] Wikipedia. *Kolmogorov–Smirnov test — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Kolmogorov%E2%80%93Smirnov%20test&oldid=1136951675. [Online; accessed 27-February-2023]. 2023.

[15] *What is numpy?* URL: https://numpy.org/doc/stable/user/whatisnumpy.html.

[16] *Scipy introduction*. URL: https://docs.scipy.org/doc/scipy/tutorial/general.html.

[17] *Visualization with python*. URL: https://matplotlib.org/.

[18] *Pandas*. URL: https://pandas.pydata.org/.

[19] *Why tensorflow*. URL: https://www.tensorflow.org/about?hl=en.

[20] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. "Enhanced Modeling and Solution of Layered Queueing Networks". In: *IEEE Transactions on Software Engineering* 35.2 (2009), pp. 148–161. DOI: 10.1109/TSE.2008.74.