Centralized and Federated Learning for Predictive VNF Autoscaling in Multi-Domain 5G Networks and Beyond

Tejas Subramanya, *Member, IEEE*, and Roberto Riggio, *Senior Member, IEEE*

***Abstract*—Network Function Virtualization (NFV) and Multi-access Edge Computing (MEC) are two technologies expected to play a vital role in 5G and beyond networks. However, adequate mechanisms are required to meet the dynamically changing network service demands to utilize the network resources optimally and also to satisfy the demanding QoS requirements. Particularly in multi-domain scenarios, the additional challenge of isolation and data privacy among domains needs to be tackled. To this end, centralized and distributed Artificial Intelligence (AI)-driven resource orchestration techniques (e.g., virtual network function (VNF) autoscaling) are foreseen as the main enabler. In this work, we propose deep learning models, both centralized and federated approaches, that can perform horizontal and vertical autoscaling in multi-domain networks. The problem of autoscaling is modelled as a time series forecasting problem that predicts the future number of VNF instances based on the expected traffic demand. We evaluate the performance of various deep learning models trained over a commercial network operator dataset and investigate the pros and cons of federated learning over centralized learning approaches. Furthermore, we introduce the AI-driven Kubernetes orchestration prototype that we implemented by leveraging our MEC platform and assess the performance of the proposed deep learning models in a practical setup.**

***Index Terms*—5G, deep learning, federated learning, autoscaling, multi-domain, multi-operator multi-access edge computing, Kubernetes.**

I. Introduction

THE 5TH generation of mobile technology (5G) and beyond (B5G) is expected to address the demands of emerging innovative applications from various vertical industries (e.g., manufacturing, automotive, healthcare, agriculture), mainly characterized by ultra-low latency, extremely high throughput, and increased connectivity density [1]. To satisfy such challenging performance requirements, Mobile Network Operators (MNOs) have considered several technological enablers, amongst which Multi-access Edge Computing (MEC) and Network Function Virtualization (NFV) are the two most critical ones [2].

MEC is an ETSI-defined network architecture concept that brings cloud computing capabilities such as compute, storage, and network to the edges of the cellular network where MEC applications could be hosted, thereby, reducing the delay experienced by users and alleviating the transport network load [3]. NFV, on the other hand, provides flexibility and programmability by enabling MEC applications to be virtualized (i.e., Virtual MEC Application Functions (VMAFs)) that could be easily deployed and scaled based on demand from end-users consuming the VMAF [4].

The Mobile Edge nodes have a limited amount of physical and virtual resource capacity compared to the cloud datacenters. Therefore, it is necessary to manage these resources efficiently. An essential characteristic of the NFV is elasticity. While NFV Management and Orchestration (MANO) entity (e.g., Kubernetes [5], Open Source MANO [6], T-nova [7]) enables VMAFs to dynamically obtain and release resources according to the varying demands, choosing the correct amount of resources is not a simple task. Current virtualization platforms offer autoscaling capabilities using a manual trigger that is reactive (e.g., if CPU utilization reaches 80%, scale-up VMAF by one). However, it would be beneficial to have a predictive autoscaling mechanism in NFV MANO that could beforehand automatically adapt the resources to the workload managed by the VMAF without any human intervention. In this article, we leverage on Deep Learning algorithms (e.g., time series forecasting) to develop predictive autoscaling solutions for effective resource utilization.

Resource scaling could be either horizontal or vertical [8]. In horizontal scaling (i.e., scaling in/out), the smallest resource unit is the VMAF (e.g., running on a container or a Virtual Machine (VM)), and new VMAFs are added or released as needed. In contrast, vertical scaling (i.e., scaling up/down) changes the resources assigned to an already running container or VM, for example, by increasing or decreasing the allocated CPU. In VMAF autoscaling, there is a trade-off between cost and Quality of Service (QoS). More VMAF instances or CPU resources need to be allocated to guarantee QoS, but allocating more resources increases the cost. Therefore, the autoscaling mechanism must be aware of the economic costs of its decisions to reduce the total expenditure but maintaining

Manuscript received May 26, 2020; revised October 28, 2020 and December 19, 2020; accepted December 23, 2020. Date of publication January 11, 2021; date of current version March 11, 2021. This work has been performed in the framework of the European Union's Horizon 2020 project 5GZORRO co-funded by the EU under grant agreement No 871533. The associate editor coordinating the review of this article and approving it for publication was T. Zinner. *(Corresponding author: Tejas Subramanya.)*

Tejas Subramanya is with the Department of Network Automation, Nokia Bell Labs, 81541 Munich, Germany (e-mail: tejas.subramanya@nokia-bell-labs.com).

Roberto Riggio is with the Department of Software Networks, i2CAT Foundation, 08034 Barcelona, Spain, and also with the Department of Connected Intelligence, RISE Research Institutes of Sweden AB, 111 21 Stockholm, Sweden (e-mail: roberto.riggio@i2cat.net).

Digital Object Identifier 10.1109/TNSM.2021.3050955

an acceptable QoS as agreed in the Service Level Agreement (SLA) between the end-user and the application provider (e.g., a round-trip-time).

Currently, industry, academia, and open-source communities (e.g., Open Network Automation Platform [9]), are focused on developing Centralized Learning (CL) algorithms for intelligent end-to-end management and orchestration of network resources. However, CL requires aggregating operational data from various data sources (e.g., MEC nodes) belonging to single or multiple domains, for insightful analytics. In the context of this article, multi-domain can either mean multiple MNO's (e.g., dealing with inter-operator Service Level Agreements) or multiple geographical domains within a single MNO. Nonetheless, such data aggregation mechanisms incur practical challenges, such as regulatory restrictions on sharing sensitive data (e.g., General Data Protection Regulation), high bandwidth resources required to transfer raw data to a central server, and increased risk associated with single point of failure. Particularly, in a multi-domain ecosystem, assuring isolation among domains is crucial because of the security concerns related to one domain possessing access to other domain's data.

Therefore, a solution is required to deal with both the multi-operator collaboration issue and address the multi-domain (i.e., different geographies) problem within a single MNO. While these two problems look different, they share a lot of familiar aspects and can benefit from having common tools to solve them. Federated Learning (FL) [10] (i.e., a distributed machine learning approach) is one such approach where data from multiple domains can be processed and analyzed in a distributed manner to perform intelligent end-to-end MANO operations. In this context, FL offers two main advantages over CL: (i) FL ensures privacy among the participating domains since the FL algorithm learns directly from the distributed domain data and only exchanges model updates among the participants. Furthermore, FL can leverage techniques such as Secure Aggregation and Differential Privacy to encrypt data to protect domains' information from other domains and hackers. (ii) FL can significantly reduce the network bandwidth because only the model parameters are sent to the centralized aggregator during training process rather than the stream of raw data like in CL.

In this article, we will leverage on FL techniques to design deep learning models for predictive VMAF autoscaling in a multi-domain setting that can better react to the changing service requirements, optimize the network resource usage, and also comply with data protection policies.

The main contributions of this article are as follows.

(i) We develop and compare different deep learning algorithms using Feed Forward Neural Networks (FFNN), Long Short Term Memory (LSTM) networks, and Convolutional Neural Networks (CNN) for predictive VMAF autoscaling, in interims of single and multi-step predictions.

(ii) We use the Federated Learning approach to develop the deep learning algorithms (i.e., those mentioned in step 1) to preserve the privacy of user data considering the multi-domain nature of 5G networks and compare their performance against the centralized approach in step 1. We investigate and evaluate the performance of two Federated Learning approaches, i.e., with and without Model Averaging, on commercial network operator dataset and compare their predictive autoscaling performance.

(iii) We present the AI-driven Kubernetes-based orchestration prototype implementation of the designed horizontal and vertical predictive autoscaling policies, based on both centralized and federated approaches, leveraging our MEC platform [11]. We then compare the benefits of a predictive autoscaling approach to reactive autoscaling.

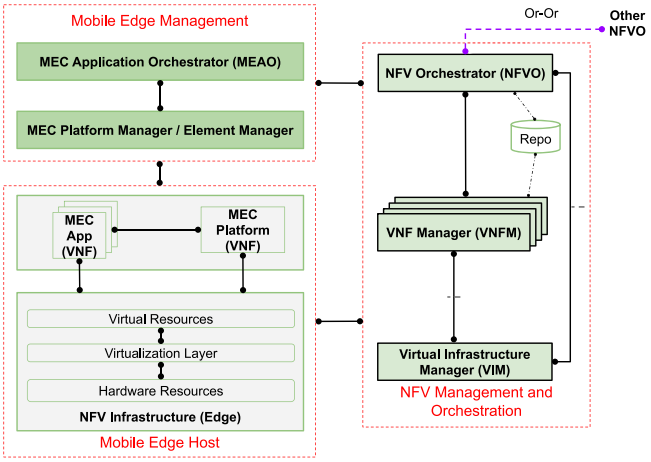


Fig. 1. MEC in NFV Reference Architecture as defined by ETSI.

II. Background

A. MANO in MEC-Enabled 5G Networks

Fig. 1 represents the MEC in NFV reference architecture, as defined by ETSI [12]. The Mobile Edge Host consists of a virtualization infrastructure that provides compute, storage, and network resources for hosting both the MEC platform and MEC applications (i.e., VMAFs). The MEC platform is a collection of essential functionality (e.g., domain name system, traffic rules manager) required to run VMAFs. The Mobile Edge Management consists of the MEC Platform Manager (MPM) and the Mobile Edge Application Orchestrator (MEAO) responsible for managing the MEC platform and the lifecycle of MEC applications. NFV MANO consists of Virtual Infrastructure Manager (VIM), VNF Manager (VNFM), and NFV Orchestrator (NFVO) responsible for managing, both physical and virtual compute, storage, and network resources, lifecycle as well and FCAPS (Fault, Configuration, Accounting, Performance, and Security) of VMAFs, and orchestrating end-to-end network services composed of multiple VNF instances. Moreover, NFVO is also responsible for exchanging of resource and service information among other NFVOs, using Or-Or interface, in a multiple administrative domain scope to guarantee an end-to-end service performance.

B. Deep Learning

Deep learning or Deep Neural Network (DNN) is a subset of machine learning in artificial intelligence that has networks capable of imitating the workings of the human brain in processing raw data and learning patterns for effective decision making. Feed Forward Neural Networks (FFNN), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN) are the three most common types of DNNs [13].

FFNN's are the simplest type of artificial neural networks that do not have any cyclic connections, and the data passes from the input layer to the output layer in a single pass without any state memory of what arrived before. A simple example of the FFNN is the Multilayer Perceptron (MLP).

CNN's are the regularized versions of MLP's that have shared-weights architecture and spatial-invariance characteristics to learn local patterns efficiently, mostly, in images. CNN's contain one or more convolutional layers, which can either be wholly interconnected or pooled. Since the convolutional layer uses a convolutional operation on the input, the network can be deeper with only a few parameters.

RNN's are networks that have cycles and include state memory to process sequences of inputs. RNN's share weights across time, unlike CNN's, which share weights across space, thus enabling them to process and represent patterns in sequential data efficiently. Most common variants of RNNs include Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU).

C. Federated Learning

Federated Learning is a recent addition to distributed machine learning, which aims at training a machine learning or deep learning algorithm, across multiple local datasets, contained in decentralized edge devices or servers holding local data samples, without exchanging their data — thus addressing critical issues such as data privacy, data security, and data access rights to heterogeneous data. This approach of Federated Learning is in contrast to traditional centralized learning techniques where all data samples are forwarded to a centralized server and also to classical distributed machine learning techniques, which assume that the local data samples are identically distributed and have the same size.

The general design of Federated Learning involves training local models on local data samples and exchanging parameters (e.g., weights in a DNN) among those local models to generate a global model. Federated Learning algorithms can use a centralized server that orchestrates the various steps of the algorithm and serves as a reference clock, or they may be peer-to-peer, where no centralized server exists [10]. In the rest of the paper (including simulations and proof-of-concept), we realize FL using a centralized server whereas the peer-to-peer FL is outside the scope of this work.

The Federated Learning process is divided into multiple rounds, each consisting of four steps:

Step 1: Local training - All local servers compute training gradients or parameters and send locally trained model parameters to the central server.

Step 2: Model aggregating - The central server performs secure aggregation of the uploaded parameters from *n* local servers without learning any local information.

Step 3: Parameters broadcasting - The central server broadcasts the aggregated parameters to the *n* local servers.

Step 4: Model updating - All local servers update their respective models with the received aggregated parameters and examines the performance of updated models.

After several local training and update exchanges between the central server and its associated local servers, it is possible to achieve a global optimal learning model.

III. Related Work

One of the significant hurdles for the deployment of NFV is the resource allocation of demanded VNFs or network services (i.e., a chain of VNFs) in NFV-based network infrastructures. A comprehensive state of the art on NFV resource allocation is discussed in [14] by presenting the fundamental research challenges and introducing a classification of the main approaches that pose solutions to solve it. One of the potential solutions for the resource allocation problem also involves VNF autoscaling [15]. Previous works on VNF autoscaling can be divided into two categories: reactive mode and predictive mode.

A. Reactive Autoscaling

In reactive mode, threshold levels can be either statically pre-defined or dynamically updated. In [16] and [17], the authors propose scalability mechanisms based on static thresholds. They define two threshold levels ($scalein_{thr}$ and $scaleout_{thr}$) to determine if the load reduces below or exceeds above the respective limits and accordingly triggers the scaling process. However, such techniques may result in oscillating behaviour affecting the overall system performance. On the other hand, [18] and [19] propose mechanisms such as queuing theory and reinforcement learning, which allows the scaling policy to be improved based on dynamic or adaptive thresholds. Although it performs better than static approaches, it remains a reactive solution with similar weaknesses.

B. Predictive Autoscaling

In predictive mode, forecasting techniques (e.g., time series forecasting) are applied to allow the systems to learn automatically and to anticipate future needs, based on which scalability decisions are taken.

The field of time series forecasting has been primarily influenced by linear statistical methods such as Moving Average (MA), Auto-Regressive (AR) and Auto-Regressive Integrated Moving Average (ARIMA). In the 1980s, it was evident that linear models do not apply to many real-world applications [20]. Therefore, researchers proposed several non-linear time series models such as the bilinear model, the threshold autoregressive model, and the Autoregressive Conditional Heteroskedasticity (ARCH). We point the readers to [20] and [21] for a detailed review of previously introduced linear and non-linear time series models.

In the last two decades, machine learning models have become main contenders to classical statistical models within

the time series forecasting community. These black-box or data-driven models are examples of nonparametric non-linear models which learn the stochastic dependency between the past and the future using only historical data. Reference [22] provides a detailed review of different machine learning models (e.g., artificial neural networks, decision trees, support vector machines, nearest neighbour regression) for time series forecasting. Furthermore, the empirical comparison of various machine learning models for time series forecasting is discussed in [23].

In the 5G domain, there already exists a lot of literature on the use of machine learning models with time series forecasting to predict traffic loads and in turn to perform VNF autoscaling. For instance, a comprehensive survey on different machine learning methods for reliable resource provisioning in edge-cloud computing ecosystem is discussed in [24]. The authors in [25] explore DNN-based and LSTM-based forecasting framework for VNF resource requirement prediction using Synthetic Minority Oversampling Technique and Batch Normalization layer to deal with the imbalanced dataset. Moreover, they explore the impact of different feature vector size and the number of hidden layers on prediction accuracy. They also propose and evaluate the performance of hybrid LSTM models such as CNN-LSTM and Bidirectional-LSTM models. Similarly, [26] offers a novel mechanism, using DNN and LSTM, to scale 5G core network virtual functions by forecasting the upcoming traffic load. Through simulations, they compare the performance of predictive VNF autoscaling to threshold-based reactive autoscaling.

Similar to the works mentioned above, in our work, we explore FFNN-based, LSTM-based and CNN-LSTM-based traffic load forecasting frameworks for VNF autoscaling. But also, we explore encoder-decoder LSTM and CNN-LSTM models for multi-step time-series forecasting in VNF autoscaling. Moreover, we consider both horizontal and vertical autoscaling, unlike other works that mostly consider a horizontal approach. But most importantly, we also propose using Federated Deep Learning techniques for predictive VNF autoscaling to preserve privacy among various participants (i.e., multi-domains) in the 5G ecosystem. Furthermore, we also present a Kubernetes-based prototype implementation of the designed autoscaling policies for both centralized and federated approaches. Additionally, we use real-operator traffic traces to generate training sets required for predicting autoscaling decisions, unlike most other works that are based on simulated datasets.

Standardization activities: Several standardization bodies such as ETSI, ITU-T, and 3GPP are currently studying the application of Machine Learning into 5G/B5G mobile network management. To name a few:

(i) ETSI Zero-touch network & Service Management [27] is focusing on a novel horizontal and vertical end-to-end architecture framework intended for closed-loop automation and optimized for data-driven machine learning algorithms.

(ii) ETSI Experiential Networked Intelligence [28] is defining a cognitive network management architecture, using Artificial Intelligence techniques to adjust provided services based on variations in user needs and business goals.

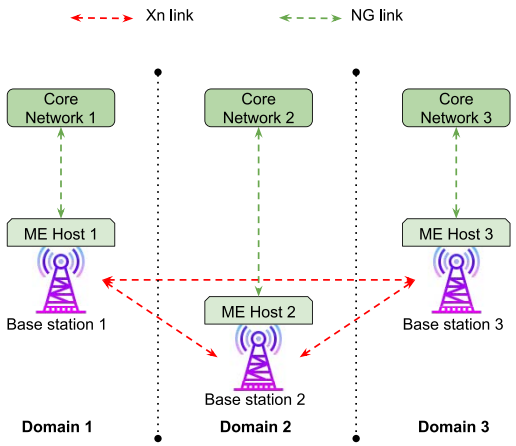


Fig. 2. Multi-domain 5G Mobile Network.

(iii) Focus Group on Machine Learning for Future Networks [29] is involved in drafting technical reports and specifications for applied machine learning in future networks, including network architectures, interfaces, protocols, algorithms, and data formats.

(iv) 3GPP specification series 37 [30] is studying about RAN-centric data collection and utilization for LTE and New Radio and also on next-generation Self-Optimizing Networks (SON) for 5G and B5G networks.

IV. Predictive Autoscaling With Centralized and Federated Deep Learning

A. Problem Statement

Consider a multi-domain 5G mobile network composed of *three* base stations that are interconnected to each other through X_n interfaces and are served by their respective core networks using *NG* interfaces, as shown in Fig. 2. Each base station in our network topology is equipped with a resource-constrained Mobile Edge (ME) Host that is capable of hosting MEC applications and exchanging/transfering data using X_n interfaces. Assume that a generic MEC application is hosted on the ME Host as a container (i.e., application code and all its dependencies are packaged into a single image), which acts as a black-box entity performing specific tasks that require computation. Suppose each MEC application has a specific performance requirement on its target response time (D_{max}) that must be satisfied. Therefore, if the incoming workload on the MEC application increases, either the amount of computing resources assigned to the application needs to be increased or multiple application instances need to be instantiated. To this end, we exploit both horizontal and vertical autoscaling of containerized MEC applications (VMAFs), which can introduce performance penalties (e.g., down-time), due to the enactment of the adaptation actions. The containerized MEC applications can be instantiated quickly and reliably using container orchestration tools (e.g., Kubernetes), further simplifying the process of managing and orchestrating resources required to run those applications.

TABLE I
STATISTICAL PROPERTIES OF TRAFFIC LOAD SAMPLES PER BASE STATION

ID	range	mean	variance	skewness	kurtosis
BS1	0 - 5.85	0.89	1.08	2.02	3.87
BS2	0 - 4.01	0.45	0.32	2.04	5.03
BS3	0 - 4.79	0.60	0.65	1.58	2.30

Stateless/Stateful VNFs: Realizing VNF scaling has been challenging, and solutions to date have not been entirely successful. The main hurdle is that many VNFs are stateful, with the state that may be read or updated quite often (e.g., per-packet, per-flow). Consequently, VNF scaling requires more than just spinning up a new VM/container and updating the load-balancer to send a portion of traffic to it. Instead, VNF scaling must also migrate states across instances and guarantee affinity between packets and their state (i.e., a packet being directed to the VNF instance that holds the state needed to process that packet). Recent works [31] have examined different options in this regard and demonstrated their advantages and disadvantages. The three proposed approaches include: (i) all state is local to the VNF, (ii) all state is remote and stored in a centralized store (also termed as stateless VNF), and (iii) a hybrid local-remote approach. In the rest of the paper, when we say VNF scaling, we refer to the remote-only approach or stateless VNF approach, which do not have to deal with state migrations.

Given Dataset: The 5G mobile network operator dataset includes historical traffic load statistics *X* of individual mobile users, covering 3 base stations, for 43 consecutive days. The traffic load samples are in the form of time series $X = \{x[t_0], x[t_1], \ldots, x[t_{i-1}], x[t_i]\}$ and the traces are collected on an hourly timescale. The samples indicate the average traffic load per second for that hour and not the peak traffic load. So, the samples do not account for any short bursts (e.g., due to unexpected events) in traffic for that hour. We observed that there exist a clear trend and seasonality (i.e., weekend spikes) in the traffic load samples for all three base stations. The randomness in the time series or the correlation among traffic load samples for all three base stations is computed, as shown in Fig. 3, through an autocorrelation plot. Due to the privacy (Non-disclosure agreement) constraints, we cannot open-source our dataset. Therefore, in Table I, we list the statistical properties of traffic load samples (in GBPS) in our dataset for each base station (BS).

Find: Firstly, centralized neural network models that can predict the future values of $x[t_i]$, i.e., the actual number of VNFs or CPU milllicores required at time $t_{i+1}, t_{i+2}, \ldots, t_{i+n}$. Secondly, neural network models that is trained in a distributed manner (e.g., using Federated Learning), where no raw data from either of the base stations are transmitted to the centralized location to preserve privacy. Finally, compare the autoscaling prediction performance of all neural network models developed.

Objective: Maximize Quality of Service (QoS) for the MEC application consumer or minimize operational cost for the service provider. The logic behind mapping the traffic load samples in *X* to the number of VNFs or 100 millicore CPU units required to process that traffic load is based on the above-chosen objective. The accuracy of the prediction largely influences the achieved goal.

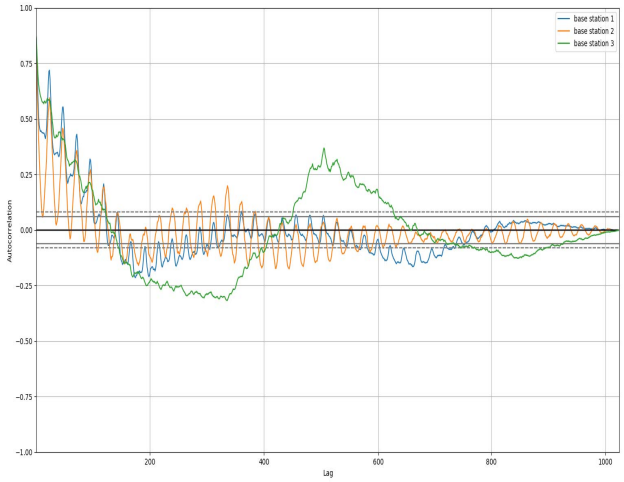


Fig. 3. Autocorrelation Plot for all three Base Stations.

B. Deep Neural Networks and Time Series Forecasting

Deep Neural Networks define parameterized functions from inputs to outputs as arrangements of many layers of fundamental building blocks called nonlinear functions. Common examples of nonlinear functions are sigmoids and rectified linear units (ReLUs). By adjusting the parameters of these nonlinear functions, such a parameterized function can be trained with the goal of fitting a given set of training samples. To be more specific, we define a loss function $L(\theta)$ over parameters θ, which is the average loss over all training samples $\{x_1, x_2, \ldots, x_n\}$. So, equation (1) represents the penalty for mismatch in the training data. The goal of training is to find the θ that results in the smallest loss, normally using the stochastic gradient descent (SGD) technique. In SGD, a batch of samples is selected in random instead of the whole data set, for each iteration, and the gradient change is computed to update θ, based on the direction of gradient change, to achieve the local minimum.

$$L(\theta) = 1/N \sum_i L(\theta, x_i) \tag{1}$$

Time is continuously moving forward, and thus, it is difficult to deal with temporal data. Time series forecasting [32] is challenging, mainly when dealing with long sequences, noisy data, multiple input/output variables, and multi-step forecasts. Deep learning techniques provide significant advantages for time series forecasting, such as automated learning of temporal dependence and automated handling of temporal structures such as trend and seasonality. In particular, LSTM neural network has been adopted mainly for time series forecasting. Adding convolutional layers to capture local, temporal patterns on top of LSTM layers can be immensely helpful in specific scenarios.

C. Data Transformation for Time Series Forecasting

In our traffic load dataset, extending backward from time *t* we have a time series of $\{x[t_i], x[t_{i-1}], \ldots, \}$. We now want to estimate *x* at a future time t_{i+s}, where *s* is called the horizon of prediction. We can predict multiple time samples in the future. This is basically a function approximation problem which is represented by equation (2).

$$x[t_{i+s}] = f(x[t_i], x[t_{i-1}], \ldots,) \tag{2}$$

We now transform this function approximation problem into a supervised learning problem by using previous time steps as input variables (X) and using the next time step as an output variable (y), and finally an algorithm is used to learn the mapping function from the input to the output. We propose two different approaches: (i) QoS prioritized time series forecasting (Y_Q) where the network operator gives priority to QoS over the cost; and (ii) Cost prioritized time series forecasting (Y_C) where the network operator chooses to neglect short-lived bursty traffic to avoid over-provisioning of VNFs or 100 millicore CPU units. Therefore, the mapping of traffic load values (*Y*) to number of VNFs or 100 millicore CPU units per VNF is based on equation (3) and (4).

$$Y_Q(t) = min\left(vnf_{max}, ceil\left(\frac{\lambda(t)}{\gamma}\right)\right) \tag{3}$$

$$Y_C(t) = min\left(vnf_{max}, floor\left(\frac{\lambda(t)}{\gamma}\right)\right) \tag{4}$$

where $\lambda(t)$ is the traffic load in a base station at time *t*, γ is the maximum traffic load a single VNF or 100 CPU millicore units can handle, and vnf_{max} is the maximum number of VNFs that can be hosted on the MEC node or (maximum CPU millicore units)/100 that can be assigned to each VNF. It is to be noted that since we do not have more details on the nature of the traffic (e.g., video or Web or audio) in our dataset, we consider the traffic to be a mix of everything. Therefore, when mapping traffic load to CPU, γ remains the same at all time intervals. For the benefit of readers, we recommend looking at [33], which describes how to determine the amount of CPU required by the VNF to process different kinds of traffic.

The successive values of our time series can be preserved if we can set up a shift register of delays (e.g., like in LSTM), which allows each past value to be an additional spatial dimension in the input to our deep learning prediction model. Since the input space to the predictor needs to be finite, at each time instant *t*, we truncate the history to only previous z samples (z is called the embedding dimension).

D. Dataset Decomposition

Our dataset is decomposed into training, validation, and test datasets. We use a rule-of-thumb decomposition conforming to 60%:20%:20% between the training, validation, and test datasets, respectively. The data samples of base station 3 do not fluctuate a lot or is almost static (also evident from the autocorrelation plot in Fig. 3). Therefore, we took a semi-randomness approach to split train/validation/test datasets by choosing (randomly) equal data samples from all three base stations. This was done to avoid having all or majority of data samples from base station 3 in training and validation datasets, which could be a possibility if we took the approach of complete random selection. If the regression performance metrics indicate overfitting, the K fold cross-validation technique can be used to generate multiple mini train-test splits to tune our neural-network models, which was not necessary in our case.

E. Centralized Learning for Deep Neural Networks

Finding the parameters of a neural-network model means searching for the best hyperparameters that can make the best predictions on the input. We applied grid search and baby-sitting as search strategies to perform an extensive search on the space of hyperparameters to find the most accurate neural-network model. This process included finding the number of hidden layers and nodes, the batch size, the regularization parameter, the learning rate of the optimizer, and the number of epochs. We encountered the process of finding hyperparameters time-consuming and hard, which assures that this topic still requires significant research. Our search space for finding optimal hyperparameters for neural-network models are as follows:

- Hidden layers (h_n) - 1 to 10.
- Nodes per layer (*d*) - 24, 48, 72 and 96.
- Activation function per layer (*A*) - ReLU and tanh.
- Optimizers (*O*) - adam and SGD.
- Learning rates (*lr*) - 0.1, 0.01 and 0.001.
- Number of epochs (*E*) - 100 to 500 in intervals of 100.
- Loss functions (*L*) - huber and mean squared error.
- Batch size (*B*) - multiples of 48 upto 480.
- Regularization - Dropout (probability of 0.2 to 0.5).

Motivation for Choosing Above Hyperparameter Range: In our dataset, in addition to weekday and weekend pattern, we also observe day and night pattern. Since our samples are in hourly timescale (i.e., 24 per day), we choose multiples of 24 (i.e., 24, 48, 72, 96) as the possible number of hidden nodes per layer. Consequently, we choose the batch size to be in the multiples of 48 (i.e., 48 to 480) to at least include two-day samples in each batch. Loss functions are chosen based on their sensitivity to outliers, i.e., mean squared error being more sensitive while Huber being less sensitive. For selecting the activation function and optimizer, we keep the most commonly used options in our search space. Finally, we would like to point out that we tried using only one hidden layer in our model, which leads to the network becoming too simple and ineffective to represent the desired mapping (i.e., high bias). Therefore, due to the increased number of hidden layers, the number of model parameters to be fitted will be much higher than our actual data sample size (i.e., 3096 samples). Consequently, we use stronger regularization methods (e.g., dropout) to prevent overfitting.

Modeling Centralized DNNs Using Tensorflow: TensorFlow [34] is an end-to-end open-source Python library for machine learning. It is characterized by a clean, uniform, and streamlined high-level API, allowing users to rapidly define, train, and evaluate machine learning models. In TensorFlow, the structure of the neural network model can be defined in a modular way using either the Sequential API

Algorithm 1: FFNN for Time Series Forecasting

$h_1, h_2, h_3 \leftarrow$ dense layer;
Dropout_Probablity $\leftarrow 0.5$;
$d_1, d_2 \leftarrow 48,\ d_3 \leftarrow 24$;
$A_1, A_2, A_3 \leftarrow$ ReLU;
$O \leftarrow$ adam, $L \leftarrow$ huber loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
train_set $\leftarrow$ *base_station*_1 + *base_station*_2 + *base_station*_3;
$model \leftarrow define(h_1, h_2, h_3, d_1, d_2, d_3, A_1, A_2, A_3)$;
for $i = 1,\ i \leq E,\ i = i + 1$ **do**
    L(*i*) $\leftarrow$ *compile*(*model*, *train_set*, *metrics*);
    new_weights(*i*) $\leftarrow$ *weight_update*(*L*(*i*), *O*, *lr*);
    update(*model*, *new_weights*(*i*))
return *model*;

Algorithm 2: LSTM for Time Series Forecasting

$h_1, h_2, h_3 \leftarrow$ LSTM layer, $h_4, h_5 \leftarrow$ dense layer;
Dropout_Probablity $\leftarrow 0.5$;
$d_1, d_2, d_3, d_4 \leftarrow 48,\ d_5 \leftarrow 24$;
$A_1, A_2, A_3, A_4, A_5 \leftarrow$ ReLU;
$O \leftarrow$ adam, $L \leftarrow$ huber loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
train_set $\leftarrow$ *base_station*_1 + *base_station*_2 + *base_station*_3;
$model \leftarrow define(h_1...h_5, d_1...d_5, A_1...A_5)$;
for $i = 1,\ i \leq E,\ i = i + 1$ **do**
    L(*i*) $\leftarrow$ *compile*(*model*, *train_set*, *metrics*);
    new_weights(*i*) $\leftarrow$ *weight_update*(*L*(*i*), *O*, *lr*);
    update(*model*, *new_weights*(*i*))
return *model*;

Algorithm 3: CNN-LSTM for Time Series Forecasting

$h_1, h_2 \leftarrow$ conv1D layer, $h_3, h_4, h_5 \leftarrow$ LSTM layer, $h_6, h_7 \leftarrow$ dense layer;
Dropout_Probablity $\leftarrow 0.5$;
$d_1, d_2, d_3, d_4, d_5, d_6 \leftarrow 48,\ d_7 \leftarrow 24$;
$A_1, A_2, A_3, A_4, A_5, A_6, A_7 \leftarrow$ ReLU;
$O \leftarrow$ adam, $L \leftarrow$ huber loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
train_set $\leftarrow$ *base_station*_1 + *base_station*_2 + *base_station*_3;
$model \leftarrow define(h_1...h_7, d_1...d_7, A_1...A_7)$;
for $i = 1,\ i \leq E,\ i = i + 1$ **do**
    L(*i*) $\leftarrow$ *compile*(*model*, *train_set*, *metrics*);
    new_weights(*i*) $\leftarrow$ *weight_update*(*L*(*i*), *O*, *lr*);
    update(*model*, *new_weights*(*i*))
return *model*;

or the Subclassing API, as a standalone and fully configurable module, which can be readily plugged together. TensorFlow offers several predefined neural layers, such as a dense layer, a recurrent layer, and a convolutional layer. A wide range of activation functions, predefined loss functions and regularization schemes are also supported.

1) Naive Time Series Forecasting: A common naive forecasting technique is to use the persistence model, which estimates the value of the next step to be the same as that of the previous step. Nevertheless, this method performs very poorly on our seasonal data. Therefore, we consider a seasonal persistence model using the observed value from the previous day, i.e., $\hat{y}_{t+i|t} = y_{t+i-q}$, where q is the seasonal period (i.e., in our case q is 24 hours or 24 samples). We will use this seasonal persistence model as the benchmark for comparison with other time series forecasting approaches.

2) FFNN for Time Series Forecasting (Algorithm 1): The time-series data, which includes one feature (i.e., number of VNFs or number of 100 millicore CPU units per VNF), is transformed into a supervised learning format with embedding dimension of $z = 48$, i.e., only 48 previous time steps are fed as input to the deep learning model. The data from the input layer is passed through 3 dense hidden layers with 48 or 24 hidden nodes per layer and using *ReLU* as the activation function in all nodes. The output layer has a single node for the regression output. For backpropagation, we use an efficient *adam* version of SGD with 0.01 learning rate, *huber* loss function, and 200 epochs. In each epoch, we calculate the loss (i.e., derivative) and then update the weight matrix for the next epoch based on the learning rate (lines 7-10). The objective is to minimize the loss on each epoch and to reach the local minimum. Once the model is defined and fit on the training data, the model can be used to predict for the next time step.

3) LSTM for Time Series Forecasting (Algorithm 2): The main difference in the LSTM model over the FFNN model is that the data from the input layer is passed through 3 LSTM hidden layers with 48 hidden nodes per layer, so as to process each input sub-sequence of 48 time steps, and using *ReLU* as the activation function in all nodes. The output of the final LSTM hidden layer is followed by 2 dense hidden layers with 48 and 24 nodes, respectively, to interpret the summary of the input sequence. The backpropagation parameters are the same as that of the FFNN model with an objective to minimize the prediction error by reaching the local minimum (lines 7-10).

4) CNN-LSTM for Time Series Forecasting (Algorithm 3): The main difference in the CNN-LSTM model over the LSTM model is that the data from the input layer is passed through 2 conv1D hidden layers, with 64 filters and a kernel size of 3, followed by a dropout layer for regularization. CNN layers can extract very informative, deep features, which are independent of time. CNN's learn very quickly, so the dropout layer is needed to slow down the learning process, eventually resulting in a better final model. The output of the final conv1D hidden layer is followed by 3 LSTM hidden layers, each with 48 nodes, and 2 dense hidden layers with 48 and 24 nodes, respectively. All hidden layer nodes use *ReLU* as their activation function. The backpropagation parameters are the same as that of the FFNN model with an objective to reach the local minimum (lines 7-10).

5) Encoder-Decoder LSTM for Multi-Step Time Series Forecasting (Algorithm 4): The Encoder-Decoder LSTM model is comprised of two sub-models: the encoder and the decoder. The encoder is responsible for reading and

Algorithm 4: Encoder-Decoder LSTM for Multi-Step Time Series Forecasting

1 $h_1, h_2, h_3 \leftarrow$ LSTM layer, $h_4, h_5 \leftarrow$ dense layer;
2 $r_1 \leftarrow$ repeatvector layer;
3 *Dropout_Probablity* $\leftarrow$ 0.5;
4 $d_1, d_2, d_3, d_4 \leftarrow 48$, $d_5 \leftarrow 24$, $d(r_1) \leftarrow 3$;
5 $A_1, A_2, A_3, A_4, A_5 \leftarrow$ ReLU;
6 $O \leftarrow$ adam, $L \leftarrow$ mse loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
7 *train_set* $\leftarrow$ *base_station*_1 + *base_station*_2 + *base_station*_3;
8 $model\_encoder \leftarrow define(h_1, r_1, d_1, d(r_1), A_1)$;
9 $model\_decoder \leftarrow define(h_2...h_5, d_2...d_5, A_2...A_5)$ **for** $i = 1$, $i \leq E$, $i = i + 1$ **do**
10 *L*(*i*) $\leftarrow$ *compile*(*model_encoder*+*model_decoder*, *train_set*, *metrics*);
11 *new_weights*(*i*) $\leftarrow$ *weight_update*(*L*(*i*), *O*, *lr*);
12 *update*(*model_encoder* + *model_decoder*, *new_weights*(*i*))
13 **return** *(model_encoder + model_decoder)*;

Algorithm 5: Encoder-Decoder CNN-LSTM for Multi-Step Time Series Forecasting

1 $h_1, h_2 \leftarrow$ conv1D layer, $h_3, h_4, h_5 \leftarrow$ LSTM layer, $h_6, h_7 \leftarrow$ dense layer;
2 $r_1 \leftarrow$ maxpooling layer, $r_2 \leftarrow$ repeatvector layer;
3 *Dropout_Probablity* $\leftarrow$ 0.5;
4 $d_1, d_2, d_3, d_4, d_5, d_6 \leftarrow 48$, $d_7 \leftarrow 24$, $d(r_1) \leftarrow 2$, $d(r_2) \leftarrow 3$;
5 $A_1, A_2, A_3, A_4, A_5, A_6, A_7 \leftarrow$ ReLU;
6 $O \leftarrow$ adam, $L \leftarrow$ mse loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
7 *train_set* $\leftarrow$ *base_station*_1 + *base_station*_2 + *base_station*_3;
8 $model\_encoder \leftarrow define(h_1, h_2, r_1, r_2, d_1, d_2, d(r_1), d(r_2), A_1, A_2)$;
9 $model\_decoder \leftarrow define(h_3...h_7, d_3...d_7, A_3...A_7)$ **for** $i = 1$, $i \leq E$, $i = i + 1$ **do**
10 *L*(*i*) $\leftarrow$ *compile*(*model_encoder*+*model_decoder*, *train_set*, *metrics*);
11 *new_weights*(*i*) $\leftarrow$ *weight_update*(*L*(*i*), *O*, *lr*);
12 *update*(*model_encoder* + *model_decoder*, *new_weights*(*i*))
13 **return** *(model_encoder + model_decoder)*;

interpreting the input sequence. The output of the encoder is a fixed-length vector that represents the model's interpretation of the time-series sequence. The output of the encoder is fed as input to the decoder.

The data from the input layer is fed to the LSTM hidden encoder layer, with 48 nodes, to read and encode the input sequences of 48 time steps. The encoded sequence is repeated three times by the model, for the 3 output time step predictions required, using a RepeatVector layer. This sequence is then passed through 2 LSTM hidden decoder layers with 48 nodes per layer. Then, the sequence is passed through 2 dense hidden decoder layers with 48 and 24 nodes, respectively, before using a dense output layer wrapped in a TimeDistributed layer to produce one output for each time step in the output sequence. All nodes use *ReLU* as the activation function. For backpropagation, we use an efficient *adam* version of SGD with 0.01 learning rate, $mean\_squared\_error$ loss function, and 500 epochs. In each epoch, we calculate the loss (i.e., derivative) and then update the weight matrix for the next epoch based on the learning rate so as to reach the local minimum (lines 9-11).

6) Encoder-Decoder CNN-LSTM for Multi-Step Time Series Forecasting (Algorithm 5): The major difference in encoder-decoder CNN-LSTM model over encoder-decoder LSTM model is that data from the input layer is passed through 2 conv1D hidden encoder layers, with 64 filters and a kernel size of 3, followed by a max-pooling layer that summarizes the most activated presence of a feature. The encoded sequence is repeated three times by the model, for the 3 output time step predictions required, using a RepeatVector layer. This sequence is then passed through 3 LSTM hidden decoder layers and 2 dense hidden decoder layers, before using a dense output layer wrapped in a TimeDistributed layer to produce one output for each time step in the output sequence. All nodes use *ReLU* as the activation function. The backpropagation parameters are the same as that of the encoder-decoder LSTM model to reach the local minimum (lines 9-11).

F. Federated Learning for Deep Neural Networks

In contrast to the centralized model training techniques, federated learning allows ME hosts belonging to multiple domains to collaboratively learn a shared global model, without the need to transfer or share raw data across domains or to a centralized server during the training process. The objective of the training is to find the best hyperparameters for the federated neural-network models that can make the best predictions on the input. Our search space for finding optimal hyperparameters is the same as that of centralized learning, which was discussed earlier. We adopt two federated learning approaches, i.e., with and without model averaging, to train the five neural-network models that were considered in the centralized approach (from Section IV-E).

Modeling Federated DNNs Using PyTorch and PySyft: PyTorch [35] is an open-source Python library that provides two high-level features: tensor computation (e.g., NumPy) and deep neural networks built on a tape-based autograd system. PyTorch also supports a rich ecosystem of tools and libraries to explore AI development. PySyft [36] is one such Python library that allows us to handle remote tensors, remote models, and remote operations through virtual or physical worker objects for secure and private deep learning using Federated Learning. In our simulations, we create 4 virtual workers, i.e., 1 for central server (e.g., a trusted third party) and 3 for ME hosts.

1) Federated Learning Without Model Averaging (Algorithm 6): Initially, we define a neural network model in the central server with the same architecture as previously used in centralized learning. We call this model *global_model* (line 6). The training process includes three steps (i.e., since we have 3 virtual ME hosts) for each epoch. First, the central

Algorithm 6: Federated Learning Without Model Averaging

1 $h_1 ... h_7 \leftarrow$ conv1D or LSTM or dense layer;
2 $d_1 ... d_7 \leftarrow$ 48or24;
3 $A_1 ... A_7 \leftarrow$ ReLU;
4 $O \leftarrow$ SGD, $L \leftarrow$ huber loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
5 *train_set*(1) $\leftarrow$ *base_station*(1), *train_set*(2) $\leftarrow$ *base_station*(2), *train_set*(3) $\leftarrow$ *base_station*(3);
6 $global\_model \leftarrow define(h_1, ..., h_7, d_1, ..., d_7, A_1, ..., A_7)$;
7 **for** $i = 1$, $i \leq E$, $i = i + 1$ **do**
8 **for** $j = 1$, $j \leq$ *no._of_ME_hosts*, $j = j + 1$ **do**
9 *model*(*j*) $\leftarrow$ *global_model*;
10 *L*(*j*) $\leftarrow$ *compile*(*model*(*j*), *train_set*(*j*), *metrics*);
11 *new_weights*(*j*) $\leftarrow$ *weight_update*(*L*(*j*), *O*, *lr*);
12 *global_model* $\leftarrow$ *update*(*model*(*j*), *new_weights*(*j*));
13 **return** *global_model*;

Algorithm 7: Federated Learning With Model Averaging

1 $h_1 ... h_7 \leftarrow$ conv1D or LSTM or dense layer;
2 $d_1 ... d_7 \leftarrow$ 48 or 24;
3 $A_1 ... A_7 \leftarrow$ ReLU;
4 $O \leftarrow$ SGD, $L \leftarrow$ huber loss, $lr \leftarrow 0.01$, $E \leftarrow 200$;
5 *train_set_*1 $\leftarrow$ *base_station_*1, *train_set_*2 $\leftarrow$ *base_station_*2, *train_set_*3 $\leftarrow$ *base_station_*3;
$global\_model \leftarrow define(h_1, ..., h_7, d_1, ..., d_7, A_1, ..., A_7)$;
6 **for** $i = 1$, $i \leq E$, $i = i + 1$ **do**
7 $model\_1, model\_2, model\_3 \leftarrow global\_model$;
8 $L_1(i) \leftarrow compile(model\_1, train\_set\_1, metrics)$;
9 $new\_weights_1(i) \leftarrow weight\_update(L_1(i), O, lr)$;
10 $L_2(i) \leftarrow compile(model\_2, train\_set\_2, metrics)$;
11 $new\_weights_2(i) \leftarrow weight\_update(L_2(i), O, lr)$;
12 $L_3(i) \leftarrow compile(model\_3, train\_set\_3, metrics)$;
13 $new\_weights_3(i) \leftarrow weight\_update(L_3(i), O, lr)$;
14 $global\_model \leftarrow federated\_avg(new\_weights_1(i), new\_weights_2(i), new\_weights_3(i))$;
15 **return** *global_model*;

server sends the *global_model* and the training configuration to the 1*st* ME host. The 1*st* ME host now trains the model on its local training dataset and updates the weight matrix based on the calculated loss. The central server now receives the updated local model (with gradients) from the 1*st* ME host and assigns it to the *global_model* (lines 8-12). Now, the central server sends the updated *global_model* to the 2*nd* ME host. The 2*nd* ME host now trains the model on its local training dataset and updates the weight matrix based on the calculated loss. The central server now receives the updated local model (with gradients) from the 2*nd* ME host and assigns it to the *global_model* (lines 8-12). The same process occurs with the $3rd$ ME host (lines 8-12). For every epoch, the same procedure is repeated until the global loss function converges, or a desirable accuracy is reached (lines 7-13).

2) Federated Learning With Model Averaging (Algorithm 7): The previous approach of Federated Learning has some significant shortcomings to guarantee data privacy. Most notably, when the central server receives the updated model from the 1*st* ME host and sends the updated model to the 2*nd* ME host, the 2*nd* ME host can look at the gradients of 1*st* ME host and restore their training data to some extent. Therefore, we employ Federated Learning with a Model Averaging approach where the gradients from multiple ME hosts are averaged before updating the *global_model*. This algorithm has two parts to it: (i) training local models on each ME host and (ii) averaging the locally trained models on the central server before updating the *global_model*.

First, the central server sends the *global_model* and the training configuration to all three ME hosts. Each ME host now trains the model on its local training dataset and updates the weight matrix based on the calculated loss. The central server now receives the updated local models (with gradients) from all three ME hosts (lines 7-13). The central server aggregates the gradients received from all three ME hosts and updates its *global_model* (lines 14). In the next epoch, the updated *global_model* is now sent to all ME hosts, and the same procedure is repeated for every epoch until the global loss function converges, or a desirable accuracy is reached (lines 6-15).

V. Kubernetes-Based Implementation

A. Docker Container and Kubernetes Orchestration

Docker [37] is a software platform designed to build, deploy, and manage applications easily, quickly, and efficiently. A Docker container image is a standalone, lightweight, executable software package, which includes the application code and all the necessary data for its execution (e.g., dependencies, system libraries and tools, configuration files). Therefore, containerized applications can be run reliably in different computing environments. Docker consists of a container-runtime (e.g., docker-engine) that enables us to create and run container images, either utilizing REST APIs or command-line interface. Docker also allows for configuring a container with a resource quota (e.g., CPU) that it can use on the host machine. The resource quota can be updated on runtime, thus also enabling vertical scaling.

Kubernetes [5] is a container-orchestration system for simplifying and automating application deployment, scaling, migration, and management across a distributed cluster of Docker-enabled nodes (e.g., Mobile Edge Host nodes, cloud nodes). A pod is the basic and smallest execution unit of an application within the Kubernetes object model that we can create/deploy. A pod comprises one or multiple application containers, storage resources, and policies on how the container must run. Kubernetes architecture follows the master-workers pattern where the master deals with the orchestration and scheduling of pods in the worker nodes based on their computational capabilities. The default scheduling policy is *Spread*, which distributes pods among all worker nodes. To scale an application horizontally, we must run multiple pods,

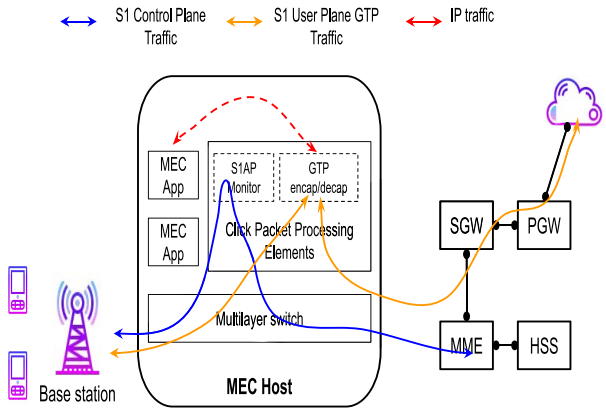


Fig. 4. A vendor-agnostic Multi-access Edge Computing prototype.

i.e., one for each instance, which, in Kubernetes, is referred to as replication.

B. A Vendor-Agnostic Multi-Access Edge Computing Platform

We introduce the vendor-agnostic Multi-access Edge Computing prototype [11], as shown in Fig. 4, which consists of an LTE base station, an LTE core network, and a mobile edge host. The mobile edge host consists of a multilayer software switch (i.e., open virtual switch), configurable packet processing elements (i.e., using click modular router [38]) for traffic routing, and containerized MEC applications (i.e., nginx webserver). We introduce two packet processing elements, i.e., S1AP Monitor and GTP encap/decap. Before jumping into the details of these elements, we need to recap on how user plane traffic is managed in the LTE Mobile Network.

Once a User Equipment (UE) completes the attach process and successfully connects to the mobile network, it can send or receive data to or from the Packet Data Networks (PDN). The uplink UE IP traffic received by the base station over the air interface is encapsulated into a GPRS Tunneling Protocol (GTP) packet before sending it to the Serving Gateway (SGW). This GTP tunnel is terminated at the SGW, where a new GTP tunnel is created towards the PDN Gateway (PGW). Finally, the PGW removes the GTP header and forwards the UE IP traffic towards the intended destination (e.g., the Internet). A similar process takes place even in the downlink direction.

For the nginx webserver MEC application to operate, the ME host has to steer IP packets from base station or core network entities towards the MEC applications running on the ME host. The MEC applications can either terminate the IP packets by itself (end-point mode) or modify the IP packets and pass it back to the original PDN connection (pass-through mode). In Bump-in-the-wire approach [39], since user plane traffic is GTP encapsulated, the S1AP Monitor element monitors the control plane messages (e.g., session management, path switch management) and extracts the UE context information (e.g., GTP tunnel IDs, UE IP address, base station and core network IP addresses). The GTP encap/decap element then performs GTP decapsulation on user plane GTP packets or re-encapsulation on IP packets based on the UE context information.

C. AI-Driven Kubernetes Orchestration in MEC-Enabled Mobile Networks

To enable predictive autonomic capabilities in Kubernetes orchestration, we extend the Kubernetes architecture to introduce the Monitor, Analyze, Plan, and Execute (MAPE) closed control loop. The *Monitor* component collects operational data (e.g., CPU), either through a centralized or distributed approach, from all deployed pods in the worker nodes. The *Analyze* component uses the collected data to perform intelligent analytics (e.g., deep learning for predictive VMAF autoscaling), either through centralized or distributed techniques, and to decide whether an adaptation is necessary for the deployed pods. If the adaptation is required, the *Plan* component defines an adaptation plan (e.g., scale VMAF1 by one instance at time $t + i$) for the deployed VMAFs, which is performed by the *Execute* component, i.e., Kubernetes master. The modularity and the support of APIs allow us to integrate our MAPE components in Kubernetes easily. To exchange data and model updates between central server (Master) and local ME hosts (Workers), we use websocket server-client connections. In practical deployments, a distributed streaming platform (e.g., Apache Kafka [40]) could be used for building real-time data pipelines instead of websocket connections.

Our AI-driven Kubernetes orchestration prototype follows the masters-workers pattern, which decentralizes the MAPE closed control loop. In particular, as shown in Fig. 5, the prototype includes one master node, which runs all the *MAPE* phases, and three ME host worker nodes, which runs either the *M* phase for centralized analytics approach or *MAP* phases for distributed analytics approach. The nodes are interconnected using a Flannel overlay network, which facilitates cluster networking. Each of these nodes runs a Kubelet, i.e., a Kubernetes node agent that plays the role of a *Monitor* component, which natively collects metrics (e.g., CPU utilization) about containers running on the node in the time-series format. The collected data is either transferred to a centralized master node or stored in the respective worker nodes. Centralized and Federated Deep Learning takes the role of *Analyze* and *Plan* components, which analyzes/interprets the collected time-series data, either through centralized or federated approaches, and predicts the required future actions. Finally, Kubernetes Master (i.e., kube-scheduler) serves the role of *Execute* component, which increases or decreases the VMAF pods (i.e., through replica sets) or 100 millicore CPU units per VMAF (i.e., by updating the CPU capacity of the pod with a new deployment) in a particular worker node. For our demonstration, nginx Web servers are containerized and deployed as MEC application pods on worker nodes.

For reactive autoscaling, the Metrics Server [41] retrieves metrics exposed by kubelet on each node through the Resource Metrics API and aggregates the cluster-wide resource usage data. The *Horizontal Pod Autoscaler* (HPA) and *Vertical Pod Autoscaler* (VPA) are custom resource definition objects (i.e.,

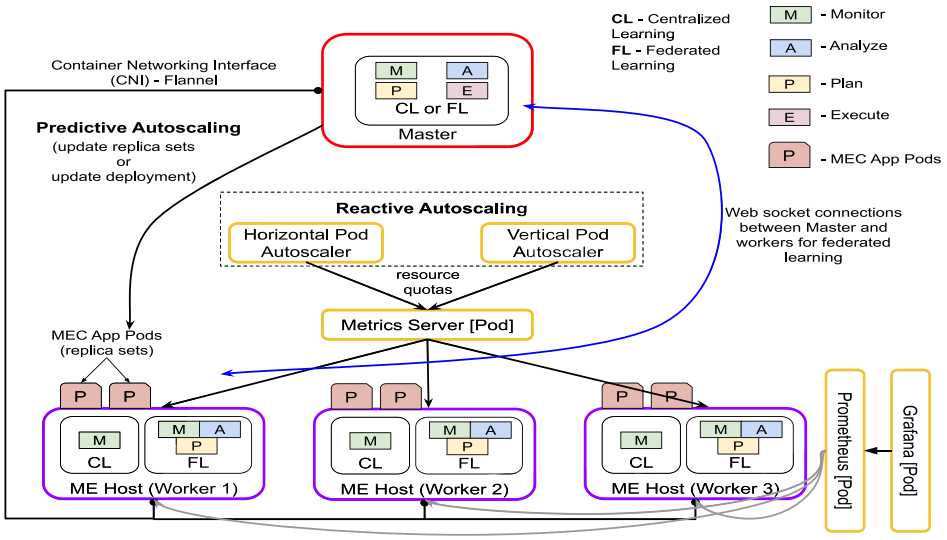


Fig. 5. Prototype of AI-driven Kubernetes Orchestration in MEC-enabled Mobile Networks.

resource quotas such as CPU, memory) for horizontal and vertical autoscaling, respectively. These autoscaler objects fetch metrics from a series of aggregated APIs provided by Metrics Server and thus facilitates autoscaling the number of pods or CPU millicores.

Moreover, the Prometheus monitoring application [42] can retrieve all the collected metrics from the worker nodes into a time-series database. It then allows querying them on-demand, thus facilitating data visualization through Grafana [43] integration.

VI. Experimental Results

A. Simulation Environment

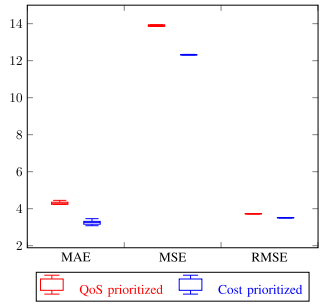
We consider a mobile network scenario with three ME Hosts, as shown in Fig. 2. Suppose each of these MEC hosts is capable of hosting MEC applications or VNFs on their NFV Infrastructure, as shown in Fig. 5. For horizontal VNF autoscaling, we assume that each ME host node can host a maximum of 10 VNFs (this assumption is based on the fact that MEC hosts are resource-constrained as compared to the centralized Cloud, which can host 100's of VNFs at a time, since they are located closer to the base station) where each of those VNFs has 1000 millicore CPU units. For vertical autoscaling, the scaling step size is set at 100 millicore CPU units (i.e., reaching up to a maximum of 1000 millicores per VNF). As traffic load increases, additional VNF/CPU instances are deployed to satisfy QoS requirements, whereas if traffic load declines, VNF/CPU instances are removed to minimize operational costs. Once the neural-network models are created as discussed earlier, the performance of these models is assessed based on correctly predicted outcomes in a test dataset. The centralized neural-network models are realized using the TensorFlow machine learning platform, while federated neural-network models are implemented using PyTorch and PySyft machine learning libraries.

We evaluate our neural-network models using three performance metrics: mean absolute error (MAE), mean squared error (MSE), and root mean squared error (RMSE). MAE calculates the average magnitude of errors in a set of predictions, without recognizing their direction (i.e., the average of the absolute values of variations between the prediction and the corresponding observation). MSE estimates the average of the squares of the errors (i.e., the average squared deviation between the predicted values and the original values). RMSE is a quadratic scoring rule which estimates the average magnitude of the error (i.e., the difference between the estimated values and the original values are each squared and then averaged over the sample). Then, the square root of the average is calculated. These performance metrics are represented using box plots, where the box portion of the box plot is defined by three lines at 25th percentile, 50th percentile and 75th percentile. The distance between the upper (75th percentile) and lower (25th percentile) lines of the box, called the inter-quartile range, gives the spread of our performance metrics.

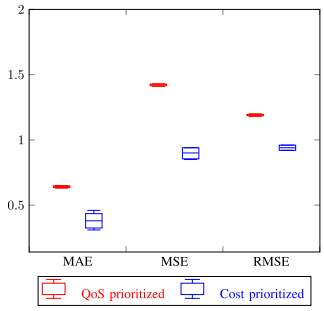
B. Simulation Results

1) Naive vs FFNN vs LSTM vs CNN-LSTM Neural-Network Models: The box plots in Fig. 6 compares the performance of Naive, FFNN, LSTM, and CNN-LSTM neural-network models for both QoS and cost prioritized horizontal autoscaling objectives by performing 10 runs of simulation for each case and measuring MAE, MSE, and RMSE in predicted number of VNF instances. Please note that we compare only the median values from the box plot. We use 3072 samples from the dataset and divide into training, validation, and test samples in the ratio of 60%:20%:20%.

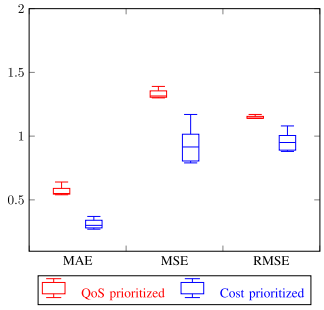
For QoS prioritized autoscaling, the CNN-LSTM neural-network model performs the best with 0.54 MAE, 1.36 MSE, and 1.16 RMSE. The second best is the LSTM model with 0.55 MAE, 1.39 MSE, and 1.17 RMSE. The third best is the



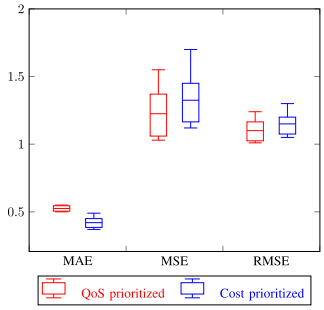
(a) Naive/Persistence model



(b) FFNN model



(c) LSTM model



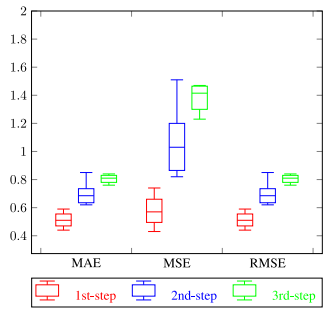
(d) CNN-LSTM model

Fig. 6. Comparision of the proposed neural-network models with QoS and Cost prioritized autoscaling objectives.

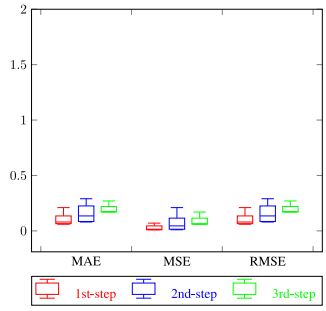
FFNN model with 0.64 MAE, 1.42 MSE, and 1.19 RMSE, and the worst performant is the Persistence model with 4.35 MAE, 13.94 MSE, and 3.73 RMSE.

For cost prioritized autoscaling, the LSTM neural-network model delivers the best at 0.31 MAE, 1.01 MSE, and 1 RMSE. The second best is the FFNN model with 0.42 MAE, 0.94 MSE, and 0.96 RMSE. The third best is the CNN-LSTM model with 0.44 MAE, 1.44 MSE, and 1.2 RMSE, and the worst performant is the Persistence model with 3.28 MAE, 12.32 MSE, and 3.51 RMSE.

2) Multi-Step Forecasting Encoder-Decoder Models: The box plots in Fig. 7 compares the performance of LSTM



(a) LSTM encoder-decoder model



(b) CNN-LSTM encoder-decoder model

Fig. 7. Comparision of the proposed encoder-decoder neural-network models for 1st-step, 2nd-step and 3rd-step predictions.

and CNN-LSTM encoder-decoder neural-network models with QoS prioritized horizontal autoscaling objective by performing 10 simulation runs for each case and measuring MAE, MSE, and RMSE for 1st-step, 2nd-step, and 3rd-step predictions of number of VNF instances.

For 1st-step prediction, the CNN-LSTM encoder-decoder model measures with 0.06 MAE, 0.01 MSE, and 0.06 RMSE, and the LSTM encoder-decoder neural-network measures with 0.52 MAE, 0.58 MSE, and 0.52 RMSE.

For 2nd-step prediction, the CNN-LSTM encoder-decoder model measures with 0.08 MAE, 0.01 MSE, and 0.08 RMSE, and the LSTM encoder-decoder neural-network measures with 0.72 MAE, 1.15 MSE, and 0.72 RMSE.

For 3rd-step prediction, the CNN-LSTM encoder-decoder model measures with 0.18 MAE, 0.07 MSE, and 0.18 RMSE, and the LSTM encoder-decoder neural-network measures with 0.82 MAE, 1.46 MSE, and 0.82 RMSE.

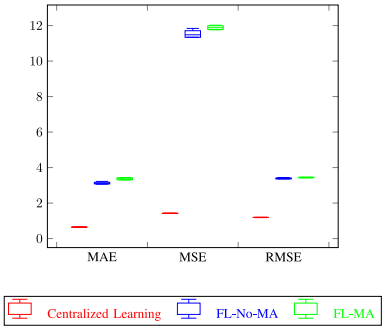
In all three cases, the CNN-LSTM model performs the best compared to the LSTM model. Also, the 1st-step predictions are better than the 2nd and 3rd-step predictions in both models.

In summary, it is worth mentioning that based on the previous analysis from Fig. 6 and Fig. 7 we can conclude that although LSTM models constitute a popular and robust choice for load forecasting time series, their usage along with convolutional layers could provide a boost in the development of an efficient forecasting model. The reason being convolutional layers in CNN-LSTM will extract useful knowledge and learn the internal representation of time-series data while the LSTM layers identify the short-term and long-term dependencies.

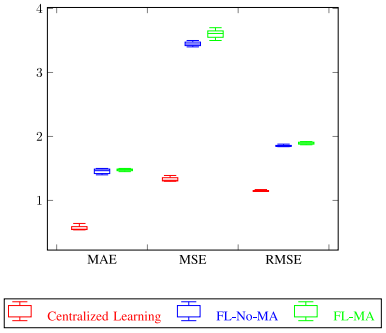
3) Centralized vs Federated Learning: The box plots in Fig 8 compares the performance of centralized learning models to federated learning models (i.e., with and without model

TABLE II
COMPARISION OF MAE, MSE, AND RMSE MEDIAN VALUES FOR CENTRALIZED AND FEDERATED NEURAL NETWORK MODELS

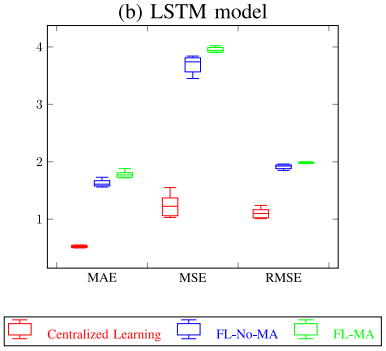
	FFNN			LSTM			CNN-LSTM		
Dataset	CL	FL - no MA	FL - MA	CL	FL - no MA	FL - MA	CL	FL - no MA	FL - MA
MAE	0.64	3.12	3.41	0.55	1.44	1.49	0.54	1.62	1.80
MSE	1.42	11.62	11.96	1.32	3.45	3.62	1.36	3.8	3.96
RMSE	1.19	3.40	3.45	1.14	1.85	1.9	1.16	1.94	1.98



(a) DNN model



(b) LSTM model



(c) CNN-LSTM model

Fig. 8. Comparision of the proposed neural-network models with centralized, federated-without-ML and federated-ML algorithms.

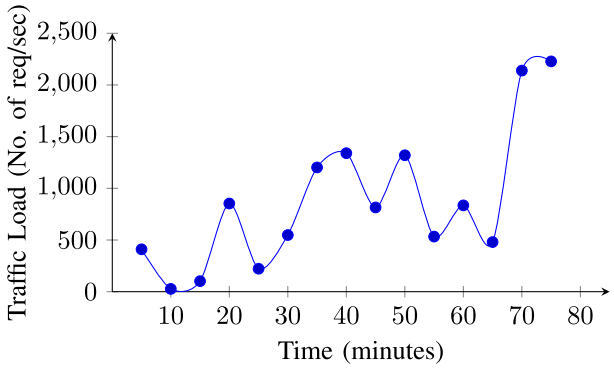


Fig. 9. Workload used in the prototype experiments.

averaging) trained on the distributed data (i.e., 3 virtual ME host nodes). The models are evaluated for QoS prioritized horizontal autoscaling objective by running 10 simulations for each case and measuring MAE, MSE, and RMSE in predicted number of VNF instances. The Table II also shows the comparison of MAE, MSE, and RMSE *median* values for centralized and federated learning neural network models. For each of the three neural network models (i.e., FFNN, LSTM, and CNN-LSTM), the centralized model performs the best. The second best is the federated learning without model averaging, and the worst performant is the federated learning with model averaging. The performance of federated learning models is poorer compared to the centralized approach, mainly because of the non-IID data distribution on each local ME host, i.e., the skewness of data distribution [44]. The skewness can be represented as the distance between the data distribution on each local ME host and the population distribution. Such a distance can be evaluated with the earth mover's distance (EMD) and is inversely proportional to the test accuracy.

C. Prototype-Based Results

Based on the encouraging simulation results, we evaluate both the centralized and federated learning algorithms in the AI-driven Kubernetes orchestration prototype that we described earlier. Furthermore, we compare the native reactive autoscaling solution against AI-driven predictive autoscaling solutions. The reference MEC application used is the Nginx Web server, which, upon request, serves the Web page request. As seen in Fig. 9, the MEC application receives a varying number of concurrent requests, such that the incoming workload pattern follows our traffic load dataset used in the simulation. However, for prototype experiments, we assume that data samples in our real operator dataset are of 5-minute granularity, thus simplifying/speeding our prototype evaluations. It is to be noted that since the traffic load samples are average values over the entire hour, we generate Web requests to achieve the same workload pattern for 5-minute intervals as well. We measure the average target response time for different approaches of horizontal and vertical autoscaling.

Here, we describe the Kubernetes configuration setting used for reactive autoscaling. For horizontal autoscaling, the HPA checks metrics values through metrics-server at 30 second time-intervals, and the relative metrics tolerance is set at 5%. The threshold levels for HPA are set at 100 millicore CPU utilization (i.e., 10% of maximum capacity) to draw parallel comparisons with vertical scaling. Additionally, the HPA waits for 3 minutes following the previous scale-up event to

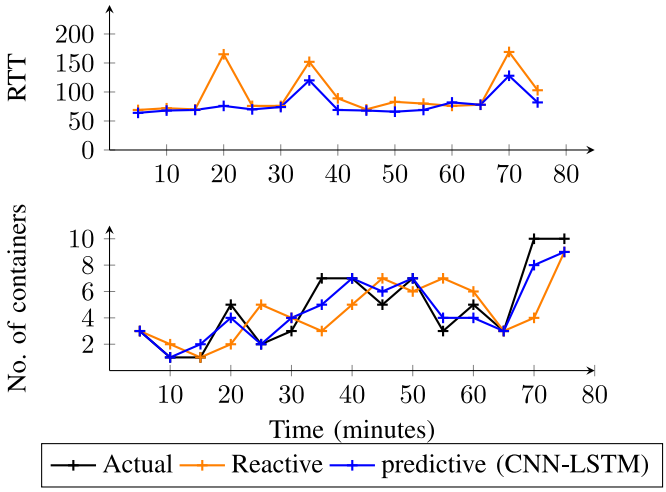


Fig. 10. Comparision of reactive and best performing predictive QoS-prioiritized centralized learning model (CNN-LSTM) for horizontal autoscaling.

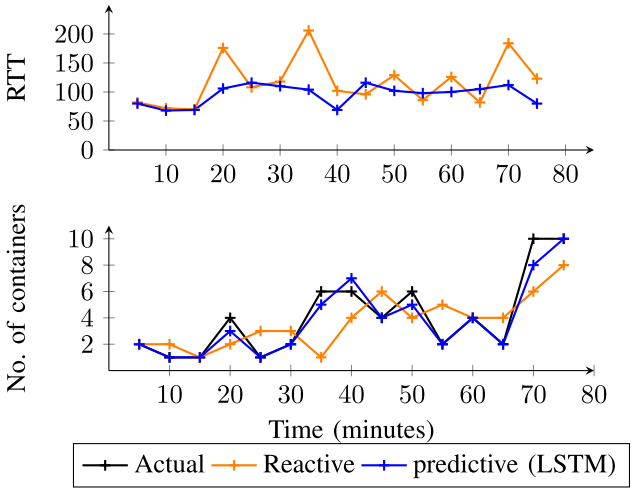


Fig. 11. Comparision of reactive and best performing predictive Cost-prioiritized centralized learning model (LSTM) for horizontal autoscaling.

allow metrics to stabilize. It then waits for 5 minutes from the last scale-down event to avoid autoscaler thrashing. For vertical autoscaling, the VPA checks metrics values through metrics-server at 10 second time-intervals and operates using *Auto* mode (i.e., not a typical reactive approach with threshold levels but uses a recommender system that can predict future pod resource requirements). The VPA waits for approximately 5 minutes following the previous scale-up or scale-down event to avoid the ping-pong effect. The changes in the resource limits of pods result in restarting the pod that might lead to instability. It is to be noted that VPA and HPA cannot work together on the same pod since both are not compatible.

1) QoS-Prioritized/Cost-Prioritized Predictive vs Reactive Horizontal Autoscaling: In the QoS-prioritized case, optimizing the average response time is more important than minimizing the resource allocation cost. Fig. 10 compares the reactive horizontal autoscaling approach with the best performing (based on our simulation results) QoS-prioritized predictive centralized learning neural network model (LSTM). Minimizing resource allocation cost is more important than optimizing the average response time in a cost-prioritized case. Fig. 11 compares the reactive horizontal autoscaling approach with the best performing (based on our simulation results) cost-prioritized predictive centralized learning neural network model (CNN-LSTM). The bottom plot represents the number of containers with respect to time (to be noted that in Figs. 10–14 black lines represent the ground-truth traffic load data at each time instance calculated using equations 3 or 4), and the upper plot represents the corresponding average round-trip-time at that particular time instance.

Based on both the figures (Fig. 10 and Fig. 11), it is evident that reactive autoscaling (i.e., orange lines) is slow in reacting to sudden traffic load spikes compared to predictive autoscaling (blue lines). The main cause being the wait time of 3-5 minutes to perform scale-up or scale-down actions by Kubernetes HPA after previous scaling events. Therefore, the reactive points in both Fig. 10 and Fig. 11 are sometimes seen above/below the predictive points, i.e., the HPA is still

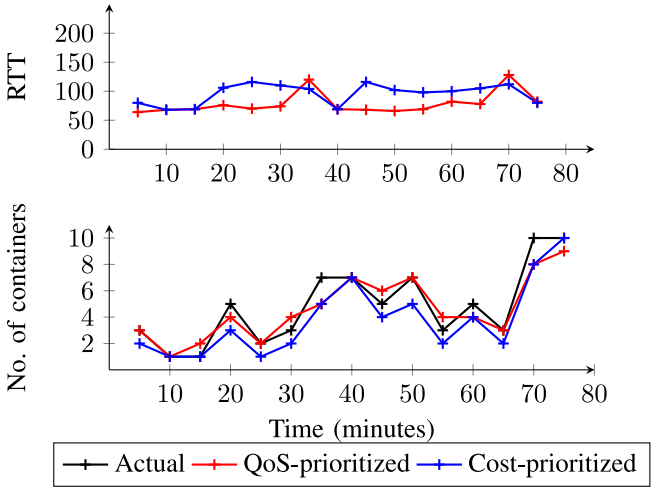


Fig. 12. Comparision of QoS-prioiritized and Cost-prioritized centralized learning model for horizontal autoscaling.

performing the scaling action of the previous timestep (3-5 minutes delayed action). Therefore, for reactive autoscaling, the number of scaled containers at a particular time instance is less/more than the actual requirement (i.e., black lines), which directly increases the overall round trip time for the MEC application users.

2) QoS vs Cost Prioritized Horizontal Autoscaling: Fig. 12 compares the performance of QoS-prioritized and Cost-prioritized predictive autoscaling approaches by plotting the predicted number of containers and the corresponding average RTTs. In cost-prioritized autoscaling (blue lines), as we see in the bottom graph, the number of scaled containers is less than that of the QoS-prioritized autoscaling (red lines) to handle the same traffic load (e.g., see x-axis at timeslot 20 min). Consequently, as we see in the top graph, the RTT (e.g., again see x-axis at timeslot 20 min) for Cost-prioritized case is much higher than the QoS-prioritized case to serve the same traffic load. However, it is to be noted that at one particular time slot (i.e., at 75 minutes in Fig. 12) the neural network for QoS-prioritized mechanism predicted 9 containers while

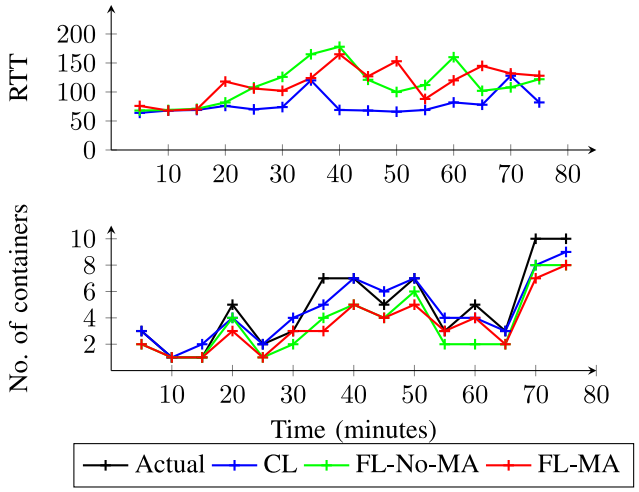


Fig. 13. Comparision of centralized, FL-No-MA and FL-MA predictive QoS-prioiritized CNN-LSTM models for horizontal autoscaling.

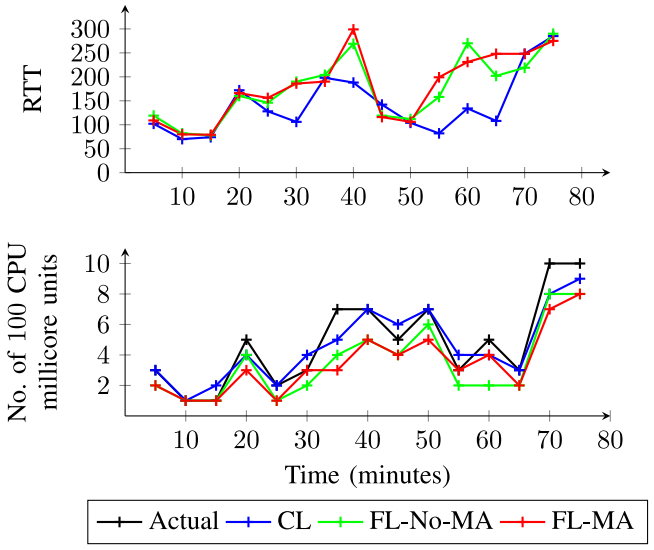


Fig. 14. Comparision of centralized, FL-No-MA and FL-MA predictive QoS-prioiritized CNN-LSTM models for vertical autoscaling.

for Cost-prioritized it predicted 10. Although we expect equal or more containers for QoS-prioritized than Cost-prioritized, we could not justify the reason for this prediction due to the black-box nature of neural networks. This would be something we plan to explore with Explainable AI in the future.

Similarly, let us consider the total cost to be directly proportional to the number of containers hosted (i.e., ignoring the cost of QoS violations). At timeslot 20 min, QoS-prioritized approach hosts 4 VNFs to achieve an RTT of 76 seconds while the cost-prioritized approach hosts 3 VNFs to achieve an RTT of 106 seconds. Therefore, the cost-prioritized approach is cheaper by 25% to QoS-prioritized approach but with a poorer QoS/RTT.

3) Centralized vs Federated Learning QoS-Prioritized Predictive Horizontal/Vertical Autoscaling: Fig. 13 and Fig. 14 compares the performance of centralized learning model to federated learning models (i.e., with and without model averaging), trained on the distributed data (i.e., 3 physical ME host nodes) using CNN-LSTM neural network, for QoS-prioritized horizontal and vertical autoscaling, respectively. The bottom plot represents the number of containers or the number of 100 millicore CPU units with respect to time. The upper plot represents the corresponding average round-trip-time at that time instance.

Based on both the figures (Fig. 13 and Fig. 14), it is clear that the centralized learning model (blue lines) performs better autoscaling predictions compared to federated learning models, i.e., with (red lines) and without (green lines) model averaging. Consequently, the overall round trip time for the MEC application users increases in the federated learning models approach. In particular, for vertical autoscaling (Fig. 14), round trip times are significantly higher compared to the horizontal autoscaling, especially at traffic load peaks, due to the pod restarts on each adaptation and the time it takes to bring the Kubernetes cluster to a stable state.

CPU Utilization: The ME hosts in our proof-of-concept experiments used Intel Core i7 processor (1.8 GHz with 4 Cores), and we observed that the CPU utilization maxes out to 100% during ML model training, thus slowing down all other processes and heating the system. Therefore, considering the resource limitations of ME hosts, running AI algorithms on ME hosts is a very challenging problem to solve. However, this is a research problem that the MEC community is trying to address over the last couple of years. In the future, MEC nodes will include specialized hardware solutions (e.g., FPGA-based GTP accelerators, GPU-based video/audio accelerators) to host low-latency MEC applications (e.g., Virtual Reality) that will make use of AI algorithms. Therefore, a percentage of such hardware resources must be reserved for 5G network management related AI algorithms. Nevertheless, we acknowledge the fact that this is a drawback in our proposed approach to use FL.

VII. Conclusion

The first part of the paper aims to design centralized and federated deep learning techniques for predictive autoscaling with QoS-prioritized and cost-prioritized objectives. We model the autoscaling problem as a time series forecasting problem and determine one-step and multi-step future predictions using real-operator traffic load datasets for training, validation, and testing. We evaluate and compare FFNN, LSTM, and CNN-LSTM models by measuring the MAE, MSE, and RMSE key performance metrics. For centralized learning and one-step predictions, CNN-LSTM performs the best for the QoS-prioritized objective and LSTM performs the best for the cost-prioritized goal. For centralized learning and multi-step predictions, the encoder-decoder CNN-LSTM model outperforms the encoder-decoder LSTM model. For federated learning, both LSTM and CNN-LSTM models perform equally better than the FFNN model. Finally, federated learning performs poorly compared to centralized learning due to the non i.i.d. data samples in the individual local ME host nodes.

The second part of the paper aims to design and implement a AI-driven Kubernetes-based orchestration prototype by leveraging our MEC platform. We evaluate both centralized and federated learning models for horizontal and vertical containerized-VMAF (i.e., Nginx webserver as a MEC

application) autoscaling by measuring the overall round trip time. Furthermore, we compare the native reactive autoscaling approach to our predictive autoscaling techniques. The prototype evaluations confirm the simulation results achieved in the first part.

REFERENCES

[1] A. Gupta and R. K. Jha, "A survey of 5G network: Architecture and emerging technologies," *IEEE Access*, vol. 3, pp. 1206–1232, 2015.

[2] I. F. Akyildiz, S. Nie, S.-C. Lin, and M. Chandrasekaran, "5G roadmap: 10 key enabling technologies," *Comput. Netw.*, vol. 106, pp. 17–48, Sep. 2016.

[3] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1657–1681, 3rd Quart., 2017.

[4] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 236–262, 1st Quart., 2015.

[5] *Kubernetes*. Accessed: May 30, 2020. [Online]. Available: https://kubernetes.io/

[6] *Open Source MANO*. Accessed: May 30, 2020. [Online]. Available: https://osm.etsi.org/

[7] M.-A. Kourtis *et al.*, "T-NOVA: An open-source mano stack for NFV infrastructures," *IEEE Trans. Netw. Service Manag.*, vol. 14, no. 3, pp. 586–602, Sep. 2017.

[8] S. Sotiriadis, N. Bessis, C. Amza, and R. Buyya, "Vertical and horizontal elasticity for dynamic virtual machine reconfiguration," *IEEE Trans. Services Comput.*, vol. 12, no. 2, pp. 319–334, Mar./Apr. 2019.

[9] *Open Network Automation Platform*. Accessed: May 30, 2020. [Online]. Available: https://www.onap.org//

[10] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 1–19, 2019.

[11] T. Subramanya, G. Baggio, and R. Riggio, "LightMEC: A vendor-agnostic platform for multi-access edge computing," in *Proc. 14th Int. Conf. Netw. Service Manag.*, Rome, Italy, 2018, pp. 198–204.

[12] *Multi-Access Edge Computing (MEC); Framework and Reference Architecture*, ETSI Standard GS MEC 003, Jan. 2019.

[13] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.

[14] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Trans. Netw. Service Manag.*, vol. 13, no. 3, pp. 518–532, Sep. 2016.

[15] H. Yu, J. Yang, and C. Fung, "Fine-grained cloud resource provisioning for virtual network function," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 3, pp. 1363–1376, Sep. 2020.

[16] S. Dutta, T. Taleb, and A. Ksentini, "QoE-aware elasticity support in cloud-native 5G systems," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2016, pp. 1–6.

[17] G. A. Carella, M. Pauls, L. Grebe, and T. Magedanz, "An extensible autoscaling engine (AE) for software-based network functions," in *Proc. IEEE Conf. Netw. Function Virtual. Softw. Defined Netw. (NFV-SDN)*, 2016, pp. 219–225.

[18] C. H. T. Arteaga, F. Rissoi, and O. M. C. Rendon, "An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an NFV-based EPC," in *Proc. 13th Int. Conf. Netw. Service Manag. (CNSM)*, 2017, pp. 1–7.

[19] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

[20] J. G. De Gooijer and R. J. Hyndman, "25 years of time series forecasting," *Int. J. Forecasting*, vol. 22, no. 3, pp. 443–473, 2006.

[21] J. G. De Gooijer and K. Kumar, "Some recent developments in non-linear time series modelling, testing, and forecasting," *Int. J. Forecasting*, vol. 8, no. 2, pp. 135–156, 1992.

[22] G. Bontempi, S. B. Taieb, and Y.-A. Le Borgne, "Machine learning strategies for time series forecasting," in *European Business Intelligence Summer School*. Heidelberg, Germany: Springer, 2012, pp. 62–77.

[23] N. K. Ahmed, A. F. Atiya, N. E. Gayar, and H. El-Shishiny, "An empirical comparison of machine learning models for time series forecasting," *Econ. Rev.*, vol. 29, nos. 5–6, pp. 594–621, 2010.

[24] T. L. Duc, R. G. Leiva, P. Casari, and P.-O. Östberg, "Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey," *ACM Comput. Surveys*, vol. 52, no. 5, pp. 1–39, 2019.

[25] Z. Zaman, S. Rahman, and M. Naznin, "Novel approaches for VNF requirement prediction using DNN and LSTM," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, 2019, pp. 1–6.

[26] I. Alawe, A. Ksentini, Y. Hadjadj-Aoul, and P. Bertin, "Improving traffic forecasting for 5G core network scalability: A machine learning approach," *IEEE Netw.*, vol. 32, no. 6, pp. 42–49, 2018.

[27] *ETSI Zero-Touch-Network-Service-Management*. Accessed: May 30, 2020. [Online]. Available: https://www.etsi.org/technologies/zero-touch-network-service-management

[28] *ETSI Experiential-Networked-Intelligence*. Accessed: May 30, 2020. [Online]. Available: https://www.etsi.org/technologies/experiential-networked-intelligence

[29] *ITU-T Focus Group on Machine Learning for Future Networks Including 5G*. Accessed: May 30, 2020. [Online]. Available: https://www.itu.int/en/ITU-T/focusgroups/ml5g/Pages/default.aspx

[30] *3GPP 37 Specification Series*. Accessed: May 30, 2020. [Online]. Available: https://www.3gpp.org/DynaReport/37-series.htm

[31] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement.*, 2018, pp. 299–312.

[32] G. P. Zhang and M. Qi, "Neural network forecasting for seasonal and trend time series," *Eur. J. Oper. Res.*, vol. 160, no. 2, pp. 501–514, 2005.

[33] H. Jmila, M. I. Khedher, and M. A. El Yacoubi, "Estimating VNF resource requirements using machine learning techniques," in *Proc. Int. Conf. Neural Inf. Process.*, 2017, pp. 883–892.

[34] *Tensorflow*. Accessed: May 30, 2020. [Online]. Available: https://www.tensorflow.org/

[35] *PyTorch*. Accessed: May 30, 2020. [Online]. Available: https://pytorch.org/

[36] *PySyft*. Accessed: May 30, 2020. [Online]. Available: https://github.com/OpenMined/PySyft

[37] *Docker*. Accessed: May 30, 2020. [Online]. Available: https://www.docker.com/

[38] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[39] "MEC deployments in 4G and evolution towards 5G," ETSI, Sophia Antipolis, France, White Paper, Feb. 2018.

[40] *Apache Kafka*. Accessed: May 30, 2020. [Online]. Available: https://kafka.apache.org/

[41] *Metrics-Server*. Accessed: May 30, 2020. [Online]. Available: https://github.com/kubernetes-sigs/metrics-server

[42] *Prometheus*. Accessed: May 30, 2020. [Online]. Available: https://prometheus.io/

[43] *Grafana*. Accessed: May 30, 2020. [Online]. Available: https://grafana.com/

[44] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated learning with non-IID data," 2018. [Online]. Available: arXiv:1806.00582.

Tejas Subramanya (Member, IEEE) received the Ph.D. degree from the University of Trento, Italy. He is a Senior Research Engineer with Nokia Bell Labs, Munich, Germany. Before that, he worked as a Researcher with Fondazione Bruno Kessler, Trento, Italy, and prior to that worked as a Research and Development Engineer with Nokia Networks, Bangalore, India. His research interests include 5G and beyond networks, network automation, and network management.

Roberto Riggio (Senior Member, IEEE) received the Ph.D. degree from the University of Trento, Italy. He is a Senior Researcher with the Connected Intelligence Group, RISE AB, Stockholm, Sweden. After that he was a Postdoctoral with the University of Florida, a Researcher/Chief Scientist with CREATE-NET, Trento, Italy, the Head of Unit with FBK, Trento, Italy, and a Senior 5G Researcher with the i2CAT Foundation in Barcelona, Spain. He has published more than 130 papers in internationally refereed journals and conferences. He has received several awards, including the IEEE INFOCOM Best Demo Award in 2013 and 2019, and the IEEE CNSM Best Paper Award in 2015. He is a member of the ACM.