Experimental Evaluation of ML Models for Dynamic VNF Autoscaling

Vasileios Zalokostas-Diplas*, Nikos Makris*†, Virgilios Passas*† and Thanasis Korakis*†
*Dept. of Electrical and Computer Engineering, University of Thessaly, Greece
†Centre for Research and Technology Hellas, CERTH, Greece
Email: vzalokost@uth.gr, nimakris@uth.gr, vipassas@uth.gr, korakis@uth.gr

2022 IEEE Conference on Standards for Communications and Networking (CSCN) | 978-1-6654-7621-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/CSCN57023.2022.10051112

***Abstract*—Network Functions Virtualization (NFV) is a key aspect deeply integrated in the latest 5G networks, allowing for the provisioning of elastic resources that adapt in a flexible manner based on the overall network demand. The adoption of NFV architectures is empowered through the evolution of cloud-native and hypervisor tools to support service monitoring, and orchestrate the appropriate decisions for provisioning the scale of the network. Such decisions may directly impact the overall quality of service and experience for users, as well as the energy consumption that the resources use. To this aim, machine learning (ML) - driven optimization for these decisions, relying on inferring the values of future monitored metrics, can assist in deciding proactively on the network scale. In this work, we employ three different candidate solutions (statistical, tree- and CNN-based) for determining the scale of network functions deployed within a cluster of resources, subject to the user demand. We compare and evaluate the different schemes in a real testbed environment, and discuss the benefits of ML-driven optimizations against existing state-of-the-art approaches.**

I. Introduction

Since their introduction, 5G networks are becoming the paradigm for the wide adoption of network softwarization, capitalizing on their unprecedented flexibility and reconfiguration. Network Functions Virtualization (NFV) has been a key technology for reducing CAPEX and OPEX cost for infrastructure owners, enhancing flexibility and scalability of the deployed functions and services. NFV, alongside with the network programmability offered by several novel interfaces (e.g. O-RAN [1], 3GPP E2 interface [2]), allow for the dynamic reconfiguration of the network, enabling the services/functions to adapt dynamically based on the network load that they are receiving. For example, network deployments might scale-in during low utilization periods or scale-out during peak hours, resulting in higher energy efficiency, rather than over-provisioning a single function for meeting the peak demand.

Nevertheless, determining the optimal point for triggering such decisions is not straightforward. They can be widely categorized in either reactive or proactive decisions, where in the former they are taken once a monitored metric value reaches a certain threshold, or in the latter case based on predicting near-future values of the metric. For the case of proactive decisions, Machine Learning (ML) approaches can be beneficial for forecasting values of monitored metrics [3], in order to reflect the future network status, based on historical patterns. Hence, the ML approach selection for forecasting network metrics can highly affect the decisions.

The research leading to these results has received funding from the European Horizon 2020 Programme for research, technological development and demonstration under Grant Agreement Number No 101008468 (H2020 SLICES-SC). The European Union and its agencies are not liable or otherwise responsible for the contents of this document; its content reflects the view of its authors only.

In this work, we attempt to shed light on the selection of the ML approach for predicting values that affect the scaling decisions for deployed VNFs. We experimentally compare three different candidate approaches for predicting monitored metrics from deployed network functions, that reveal the true network resource demand. The approaches that we evaluate are based on the state-of-the-art algorithms for either statistical based models i.e. *ARIMA*, tree-based approaches with the *XGBoost* method, or CNN based models with *LSTM*. We target the scaling process that is integrated in orchestration tools for cloud-native functions. In this work, we extend the scaling capabilities offered by the Kubernetes (K8s) orchestrator, towards proactively determining the network resource demand in the near future, and appropriately scale the deployed network functions. The approach can easily extend to several other virtualized services that can be managed in a similar manner, e.g. the service-based 5G Core Network. Our approach is evaluated experimentally in a cluster of nodes, while the network resource demand is emulated using an open-source dataset [4] which includes traffic from real network users.

The rest of the paper is organized as follows. In Section II we provide details on relevant literature. In Section III we describe the system model and our applied optimizations in the ML modes. In Section IV we provide our experimental findings, and in Section V we conclude our work and present our future directions.

II. Related Work

The transition of the networks to use virtualization, through the execution of each dedicated function as a VNF, has added up to the overall flexibility for network management. This has allowed several novel approaches to emerge, allowing the network to scale [5] based on monitored target metrics. The most widely used approach is the implementation by the Kubernetes orchestrator, which is able through the Horizontal Pod Autoscaler, to spawn multiple replicas of the same service to meet the demand. Nevetheless, this process depends on monitored metrics, and is a reactive solution, spawning the replicas as soon as a target metric is reached.

As the data monitored from such tools usually include time as a parameter, this creates space for the application

Authorized licensed use limited to: University of Thessaly. Downloaded on December 08,2023 at 11:14:48 UTC from IEEE Xplore. Restrictions apply.

of Machine Learning (ML) approaches towards predicting their future values [3] or network malfunctions [6]. Several works take advantage of this, and present their own approach. For example, in [7] the authors present their own autoscaling algorithm that tries to balance the tradeoff between performance and operational costs. Their results show that the proposed algorithm reduces operation costs, bound to the maximum latency allowed for the end users accessing the deployed services. In [8], authors evaluate through simulations several deep learning models, both centralized and federated approaches, that can perform horizontal and vertical autoscaling in multi-domain networks. They model the autoscaling problem as a time series forecasting problem that predicts the future number of VNF instances based on the expected traffic demand. Their contributions feature the evaluation of Feed Forward Neural Networks, Long Short Term Memory (LSTM) networks, and Convolutional Neural Networks - LSTM (CNN-LSTM). Authors in [9] compare three different types of Deep Reinforcement Learning in order to determine when their functions should be scaled. They propose and compare a Deep Reinforcement Learning (DRL) agent, a classical Proportional-Integral-Derivative (PID) controller, and a Threshold (THD)-based algorithm for determining the amount of VNF instances to fulfill service latency requirements without knowing or predicting the expected demand. Similarly, in [10], authors investigate the use of ML techniques to estimate VNFs needs in term of CPU as a function of the traffic they will process. In this work, they use a Support Vector Regression (SVR) approach. In work [11], authors develop their own scheme for flow migration, when autoscaling the deployed functions.

Other approaches in related work include [12], where ML is used for resolving the placement problem for VNFs. The authors use a Deep Deterministic Policy Gradient Reinforcement Learning algorithm, to fully automate the Virtual Network Functions deployment process between edge and cloud network nodes. In [13] authors present a ML methodology for generating labeled training data that reflects temporal dynamics on a deployed network, and determines the optimal topology in WAN and mobile edge computing environments. Finally in [14], authors use a virtual network function-forwarding graph (VNF-FG) to combat the problem of dynamic VNF allocation, while considering VNF migration.

In this work, we extend the scaling capabilities offered by the K8s orchestrator, towards proactively determining the network resource demand in the near future, by employing three different ML approaches. Our goal is to determine which one of them performs better for our case, and to eventually scale the deployed network functions towards ensuring that the overall network will be energy efficient.

III. System Architecture and Model

As the base of our development, we use the Kubernetes orchestrator, that allows scaling of the deployed network functions. Kubernetes (K8s) [15] is part of the Cloud Native Computing Foundation (CNCF) which supports the development of shared networking standards in cloud data management software. It is an open-source system for automating deployment, scaling and management of containerized applications and groups of containers that make up an application into logical units for easy management and discovery.

K8s empowers high availability and scalability through various automatic scaling mechanisms. Autoscaling allows the cluster to dynamically adjust to demand without the intervention from individuals in charge of operating the cluster. Without it, the developers must manually provision resources every time conditions change, and it is less likely to be operating with optimal resource utilization and cloud spending. Three types of autoscaling are supported in the K8s ecosystem:

- *Cluster Autoscaler (CA)* that increases or decreases the size of a cluster by simply adding or removing nodes.
- *Horizontal Pod Autoscaler (HPA)* that automatically scales up/down the number of resource units, called Pods.
- *Vertical Pod Autoscaler (VPA)* that dynamically modifies the attributed resources like CPU and RAM of each node in the cluster.

In this work, we focus on the horizontal scaling case, and compare our solution with the off-the-shelf K8s HPA.

A crucial part for managing the deployed workloads is monitoring of different metrics from the deployed functions that reflect the true load under which the functions are placed. Towards integrating our proposed approach with existing functionalities provided by K8s, we adopt the *Prometheus* monitoring and alerting toolkit [16]. Prometheus collects and stores metrics from either host nodes (bare metal) or deployed workloads (containers) as time series data i.e., metrics are stored with a timestamp of the recorded time, alongside optional key-value pairs. Prometheus integrates with K8s through the *prometheus-adapter* extension, able to serve metrics through the K8s API. Therefore, Prometheus collected metrics can be leveraged for several operations within K8s, such as scaling decisions for the autoscaler.

The off-the-shelf autoscaler uses default memory or CPU consumption metrics, that upon reaching a certain target metric, the scaling process is triggered. Through Prometheus and the respective adapter, the scaling process can extend to other monitored metrics. For example, in the case of a web server, incoming requests can be appropriately monitored, and scaling decisions can be taken using them as target metrics.

In this paper, we are focusing on the Horizontal's Pods Autoscaler functionality which scales the number of pods based on a custom specific metric. The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. We extend the autoscaler functionality accordingly, in order to forecast the evolution of the monitored metrics in the future, and appropriately scale the deployment towards ensuring energy savings for the infrastructure. The developed controller adjusts the number of replicas (identical instances of a pod) based on the observed metrics to the target specified by the developers. Metrics are fetched using the Kubernetes resource metrics API or the custom metrics API. The second type of API was used since the Prometheus metrics are served

Authorized licensed use limited to: University of Thessaly. Downloaded on December 08,2023 at 11:14:48 UTC from IEEE Xplore. Restrictions apply.

over it. In this manner, our solution is pluggable to any similar deployment, by changing the monitored service and the respective decision metrics.

A. Dataset and target metrics

For our setup, we use a target web service deployed in our K8s cluster, and generate traffic accordingly in order to scale it depending on the demand. Towards achieving this, the Prometheus tool is configured to monitor the incoming requests to the web server. This metric is used as the scaling decision in our experiments, for the off-the-shelf Kubernetes autoscaler and our own ML driven. In order to generate the requests to the web service, we use an online available dataset [4] that contains all the connections served over 4G base stations in the Milan area over the course of the week. We isolate the traffic from two base station (the most busy and a medium utilized one) and replay the requests to the web service. In the following section we detail our ML approaches for predicting the evolution of this specific metric.

B. Machine Learning approaches

In this subsection, we present the Machine Learning (ML) approaches that we followed in order to determine the most appropriate solution for forecasting our monitored metrics. For all the approaches, we used a 5-step out-of-sample prediction mechanism. This consists of the following: 1) at first, every model uses a dataset for input, in an appropriate format, used for training. 2) Subsequently, the ML model is being fitted on the training data so that it can be used for forecasting. 3) The forecast that is being produced is just an observation that is not part of the input data and that's why it is called out-of-sample. 4) Now, in order to make a 5-step out-of-sample forecast, every prediction that is being made, is used as input for the next one and the entire process is repeated 5 times. To make things clearer, an example of a 3-step prediction is being presented in Figure 1.

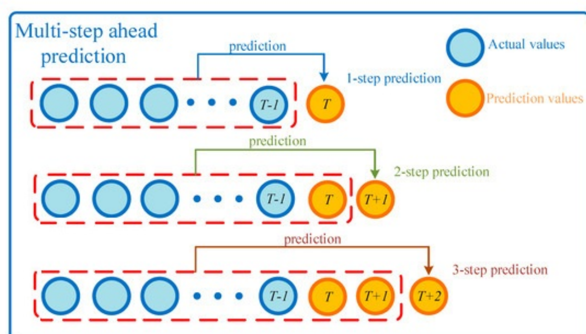


Fig. 1: Multi-step prediction employed in this work

As our ML model, we used and evaluated on their accuracy three different ML, from three different categories as follows:

- Classical/Statistical models that have mainly strong base in statistics like Moving Averages and Exponential Smoothing. The AutoRegressive Integrated Moving Average (ARIMA) model [17] was used from this category.
- Machine Learning reduction models, such as Random Forests. The XGBoost [18] algorithm was used from this category.
- Deep Learning models for analyzing data with a time component. The Long Short Term Memory (LSTM) [19] model was used.

In the following subsections, we present some brief details for each one of them.

1) AutoRegressive Integrated Moving Average - ARIMA: ARIMA as a statistical based model, features Autoregression (AR) for specifying the dependent relationship between an observation and lagged observations. It also differences the raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to ensure that the data included in the model are stationary (Integrated - I). Finally, it uses the dependency between observations and a residual error from a moving average model in order to apply it to lagged observations. All these components can be specified in the model as parameters. Subsequently, a liner regression model is constructed.

2) XGBoost: XGBoost is an open-source software library standing or eXtreme Gradient Boosting. It is a gradient boosting decision-tree based algorithm. XGBoost has gained a lot of attention lately, due to its computational speed and model performance. Unlike ARIMA, the input data must be prepared accordingly for XGBoost in order to transform the problem to a supervised learning one. Supervised learning is an approach to Machine Learning where the machine learns from labeled data. So, samples that have not seen before by the learner are fed to the model and a prediction is made based on the mapping learned.

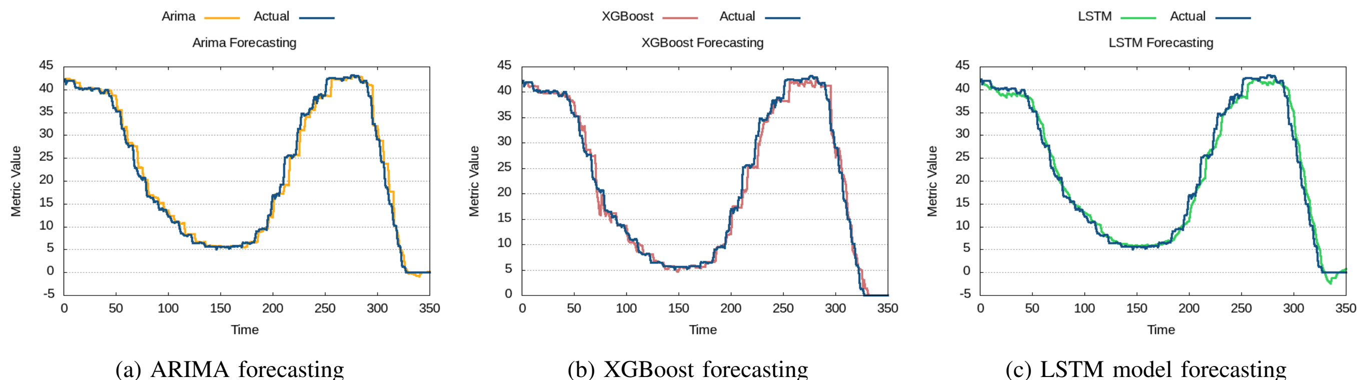
3) Long Short-Term Memory - LSTM: The Long Short-Term Memory (LSTM) network is a type of Recurrent Neural Network used in deep learning. Recurrent networks have an internal state that represents context information and keeps track of the past inputs. LSTM models can identify and handle long-term dependencies by using feedback connections and not forward-feeding. They use memory blocks with gates that manage the state and output of each component. Similar to XGBoost, the data needs to be processed before feeding them to the model; data needs to be preprocessed into multiple input/output patterns (samples) where each input sequence is used to produce a single output.

C. ML model selection and evaluation

In this section, we present a first evaluation that took place in order to determine the effectiveness of each model in forecasting the different values in the system. The evaluation is driven by the tuning of all the hyperparameters for each model. The following subsections present the specific parameters used for each model in our architecture.

1) ARIMA Parameters: In order to evaluate the accuracy of the ARIMA model, as well as to have a first view on how it would perform in the real-time scenario that the scaling mechanism will use to execute it, the dataset (see sect. III-A) containing the connections served by the deployed service was

Authorized licensed use limited to: University of Thessaly. Downloaded on December 08,2023 at 11:14:48 UTC from IEEE Xplore. Restrictions apply.



(a) ARIMA forecasting (b) XGBoost forecasting (c) LSTM model forecasting

Fig. 2: Evaluation for ARIMA, XGBoost and LSTM performance

split into two parts, one for training the model, and one for testing its accuracy. The initial dataset, contains the number of connections that get served over the network, and consists of 1412 points in total. Out of these, the last 350 are used as the test part, and the rest as the training set. The model is set to forecast the next five values in each step. These values are averaged in the output forecasting, in order to minimize the produced error (as it is further presented in the next subsections). The results on the comparison of the actual values with the forecasted values are shown in Figure 2a.

2) XGBoost Parameters: Due to its decision-tree based approach, XGBoost was found to perform better when using about 50 points as input. Two hyperparameters values were chosen: *objective* and *n_estimators*. Objective specifies the learning task and the corresponding learning objective to be used. A wide variety of objectives were tested such as *count:poisson*, *reg:gamma*, *reg:squarederror*, with *reg:tweedie* being the best one for this case, due to the lowest Mean Average Error. As far as the *n_estimators variable*, it represents the number of gradient boosted trees. By evaluating the mean average error, we concluded in configuring it to a value of 20 for the XGBoostRegressor function. The same dataset scenario as in the ARIMA case was used in order to test this algorithm's accuracy. The dataset was split into train and test in the same manner as in ARIMA. Figure 2b shows the XGBoost performance over the test data.

3) LSTM Parameters: As in the previous cases, the LSTM model had to be tuned accordingly. The hyperparameters that had to be tuned are: 1) **LSTM Units**: refers to the number of units of the LSTM network. Using a higher number of units indicates a more powerful/precise network, but raises the training time, with the possibility to overfit the data. For our case, 400 LSTM units was found to produce accurate forecasts. 2) **Number of epochs**: is the number of times that the learning algorithm will work though the entire training dataset. For our model, we used 40 epochs of training time. 3) **Loss function**: is the function that calculates the error used in the training process. For our model, we use the Mean Squared Error (MSE) loss function, which calculates the loss based on the difference between the model's predictions and the ground truth, squaring it and averaging it across the whole dataset. 4) **Optimizer**: is a method for changing attributes (weights/learning rate) of the neural network in order to reduce the losses. The "Adam" optimizer [20] was used because its effectiveness.

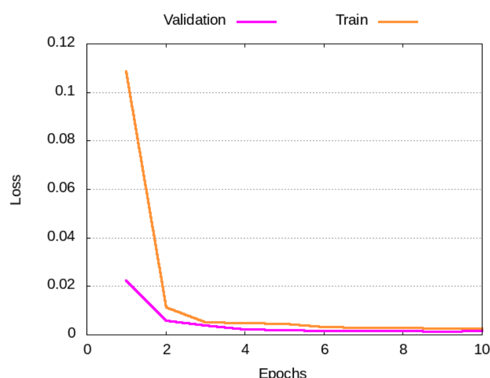


Fig. 3: Data fitting for the under-test LSTM model

In order to make forecasts and validate and visualize the results as with the ARIMA and XGBoost cases, the same methodology was used. The input dataset was split into train and test. Figure 2c shows the LSTM network performance over the same test dataset as the other algorithms. For such neural networks, it can be difficult to diagnose that the model is performing as expected. A good skill score might be extracted but it is important to know whether or not the model is a good fit for the training data or if it is under/over-fitting. A good fit is identified by a training and validation loss and specifically, if both of them decrease to a point of stability and the validation loss has a small gap with the training loss. Figure 3 shows that our models converges to low losses in about 9 epochs of training time, and thus has a good fit to our data. Although training seems to reach a near to minimum value for the loss function in about 3 training epochs, our results with less than 9 training epochs yielded significantly higher Mean Absolute Error (MAE) evaluation in the final testing data.

D. Mean Absolute Error Evaluation

In order to better visualize the differences in accuracy of the different solutions, we also use the Mean Average Error (MAE). MAE can reveal how accurate a forecast system is, by measuring this accuracy as a percentage, and can be calculated

Authorized licensed use limited to: University of Thessaly. Downloaded on December 08,2023 at 11:14:48 UTC from IEEE Xplore. Restrictions apply.

as the average absolute percent error for each time period minus actual values divided by actual values. Table I shows the MAE of every algorithm when using the same test dataset. As it is clear, the statistical method is performing a better than the tree- or LSTM approach. This is happening due to our processing for each prediction round, as we forecast the next five values, and average them as our predicted value.

TABLE I: Mean Average Error evaluation

Mean Average Error		
Arima	XGBoost	LSTM
1.36	3.15	3.67

IV. System Setup and Evaluation

For the experimental evaluation of our ML-based real-time scaling mechanism we used NITOS Testbed. NITOS [21] is an integrated facility with heterogeneous testbeds that focuses on supporting experimentation-based research in the area of wired and wireless networks located in the city of Volos, Greece.

For our experimental setup, we target on applying the ML approaches for the Kubernetes autoscaler. The autoscaler is managing a web service (realized with php and Apache2) deployed on the cluster, able to receive and serve requests to the end users. We implemented a mechanism inside the Prometheus tool, that monitors this deployment and collects resource data that describe the web requests arriving at the web server. The data are used to train the ML approaches, and based on their forecasted values, we choose to proactively scale the deployment. In this way, resources are being allocated or released based on future traffic, so that they are prepared to respond faster and more efficiently while at the same time saving energy as there are no resources that overwork or are idle. Also, this custom scaling mechanism runs in real-time by appending the resource data to the dataset that is used to train our algorithms periodically (per 3 minutes) so that the ML predictions would be as close to the reality as possible. For our evaluation, we compare our solution with the off-the-shelf K8s Horizontal Pod Autoscaler.

We used 2 NITOS nodes that operated as a Kubernetes Master and a Kubernetes Worker. We defined a deployment that utilizes a php-apache server able to receive HTTP requests from the outgress of the cluster, and subsequently generated the requests towards the service using the aforementioned dataset. We use two scenarios of traffic from the dataset: 1) the one with the most requests (up to 14K requests within an hour) and 2) one with a medium profile of traffic (approx. up to 8K requests within the most busy hour of the day). In order to test our ML mechanisms, we used the traffic patterns from the high traffic dataset for training our algorithms, and tested them with unseen data from the medium traffic dataset. The two datasets are visualized in Figure 4.

In Figure 5 we demonstrate our results with respect to the number of replicas active in the system. We compare our results with the default K8s HPA, that is triggered once the monitored metric (number of requests arriving at the monitored

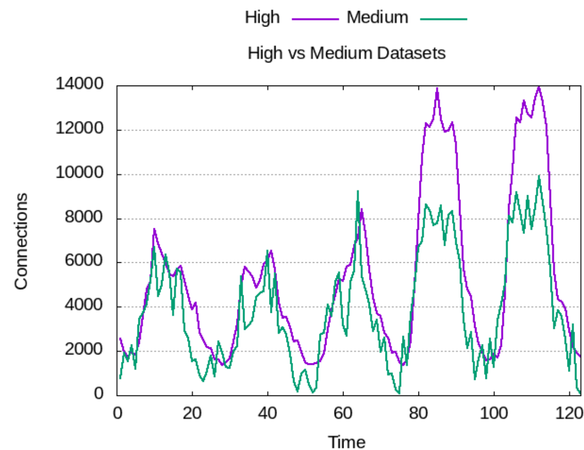


Fig. 4: The two traffic datasets used in our experiments

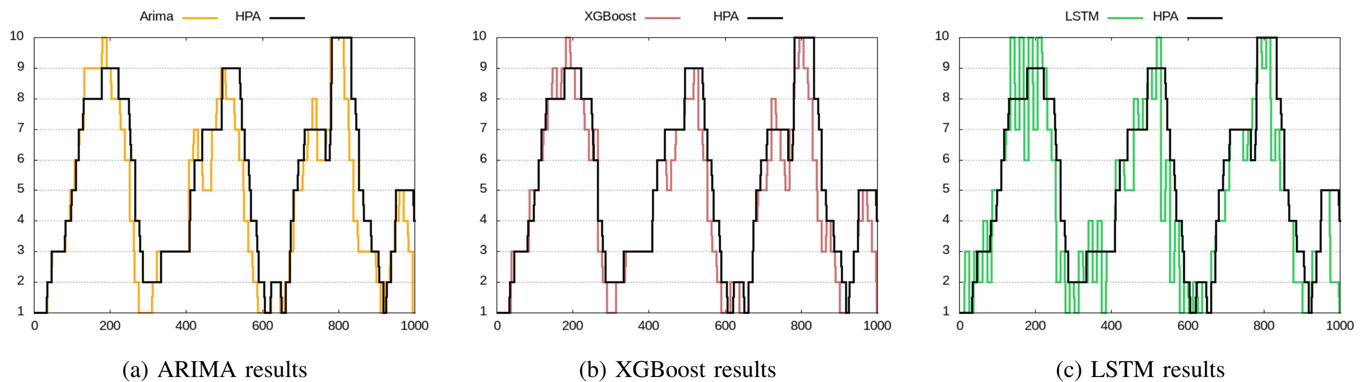
service) reach a target. As it can be seen, for almost all cases, a better use of resources is achieved as the scaling is done earlier using this mechanism. More specifically, the same number of replicas is being scaled several seconds before the HPA would do, on most of the occasions.

In Figure 6, we illustrate the total CPU utilization for every experiment that took place. As the scaling process happens earlier than the HPA, better use of resources is being expected to appear. We predict the upcoming traffic in order to scale in time and when the traffic arrives everything is scaled as they should do, so that there are no resources that overwork or being allocated and not serving the traffic. That is why better CPU utilization is being achieved implementing our scaling mechanism even with the LSTM implementation which produced the least accurate predictions in comparison with XGBoost and Arima. Projecting the CPU utilization to energy efficiency, by calculating the area below the CPU utilization plots, we conclude that the ARIMA solution is creating the most energy-efficient results, due to its ability to better predict the upcoming requests for our dataset.

V. Conclusion and Future Work

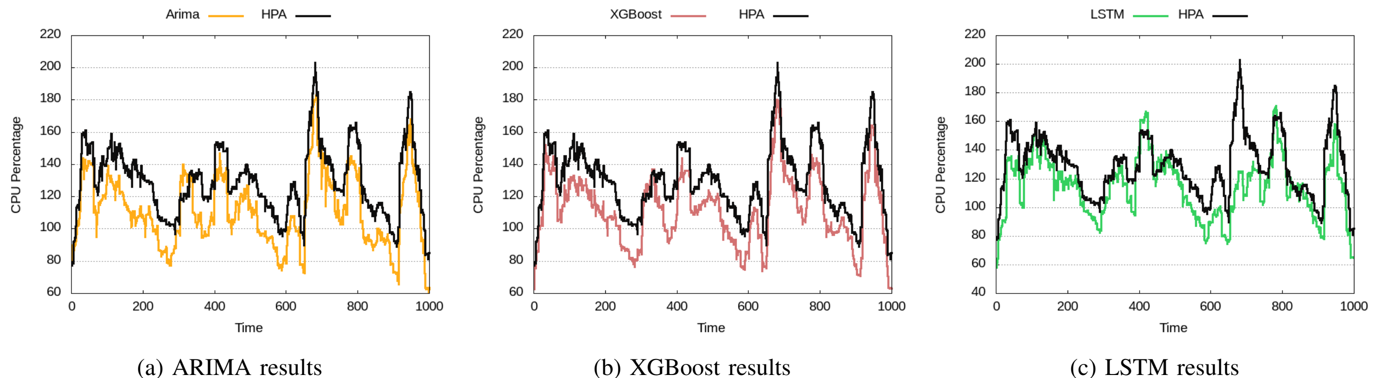
In this work, we experimentally evaluated three different candidate solutions (statistical, tree- and CNN-based) for determining the scale of network functions deployed within a cluster of resources. All three solutions are found to be outperforming the off-the-shelf scaling solutions that exist in the K8s orchestration environment. Out of the three solution, the statistical based one (ARIMA) seems to have better suit our data, and hence present more accurate predictions. By projecting our results to the total CPU utilization (which in turn reveals the energy efficiency), we are able to conclude that our proposed approach can lead to higher efficiency for systems that employ such a pro-active approach. In the future, we foresee to extend our schemes to manage specific functions of the Telecom network (e.g. the 5G Core Network, or parts of the RAN that are executed as functions in the cloud).

Authorized licensed use limited to: University of Thessaly. Downloaded on December 08,2023 at 11:14:48 UTC from IEEE Xplore. Restrictions apply.



(a) ARIMA results (b) XGBoost results (c) LSTM results

Fig. 5: Experimental results on number of replicas for each algorithm vs the K8s HPA



(a) ARIMA results (b) XGBoost results (c) LSTM results

Fig. 6: Experimental results on CPU Utilization for each algorithm vs the K8s HPA

References

[1] A. Garcia-Saavedra and X. Costa-Pérez, "O-RAN: Disrupting the Virtualized RAN Ecosystem," *IEEE Communications Standards Magazine*, vol. 5, no. 4, pp. 96–103, 2021.

[2] S. Kukliński, L. Tomaszewski, and R. Kołakowski, "On O-RAN, MEC, SON and network slicing integration," in *2020 IEEE Globecom Workshops (GC Wkshps*. IEEE, 2020, pp. 1–6.

[3] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, "Auto-Scaling VNFs Using Machine Learning to Improve QoS and Reduce Cost," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.

[4] G. Barlacchi, M. De Nadai, R. Larcher, A. Casella, C. Chitic, G. Torrisi, F. Antonelli, A. Vespignani, A. Pentland, and B. Lepri, "A multi-source dataset of urban life in the city of Milan and the Province of Trentino," *Scientific data*, vol. 2, no. 1, pp. 1–15, 2015.

[5] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–5.

[6] I. Syrigos, N. Sakellariou, S. Keranidis, and T. Korakis, "On the Employment of Machine Learning Techniques for Troubleshooting WiFi Networks," in *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*, 2019, pp. 1–6.

[7] Y. Ren, T. Phung-Duc, J.-C. Chen, and Z.-W. Yu, "Dynamic Auto Scaling Algorithm (DASA) for 5G Mobile Networks," in *2016 IEEE Global Communications Conference (GLOBECOM)*, 2016, pp. 1–6.

[8] T. Subramanya and R. Riggio, "Centralized and Federated Learning for Predictive VNF Autoscaling in Multi-Domain 5G Networks and Beyond," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 63–78, 2021.

[9] P. Soto, D. De Vleeschauwer, M. Camelo, Y. De Bock, K. De Schepper, C.-Y. Chang, P. Hellinckx, J. F. Botero, and S. Latré, "Towards Autonomous VNF Auto-scaling using Deep Reinforcement Learning," in *2021 Eighth International Conference on Software Defined Systems (SDS)*, 2021, pp. 01–08.

[10] H. Jmila, M. Ibn Khedher, and M. El Yacoubi, "Estimating VNF resource requirements using machine learning techniques," in *ICONIP 2017 : 24th International Conference on Neural Information Processing*. Guangzhou, China: Springer, Nov. 2017, pp. 883 – 892. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01682996

[11] X. T. Vu, J. Lee, Q. H. Nguyen, K. Sun, and Y. Kim, "An architecture for enabling VNF auto-scaling with flow migration," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, 2020, pp. 624–627.

[12] A. Dalgkitsis, P.-V. Mekikis, A. Antonopoulos, G. Kormentzas, and C. Verikoukis, "Dynamic Resource Aware VNF Placement with Deep Reinforcement Learning for 5G Networks," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–6.

[13] S. Lange, H.-G. Kim, S.-Y. Jeong, H. Choi, J.-H. Yoo, and J. W.-K. Hong, "Predicting VNF Deployment Decisions under Dynamically Changing Network Conditions," in *2019 CNSM*, 2019, pp. 1–9.

[14] P. T. A. Quang, A. Bradai, K. D. Singh, G. Picard, and R. Riggio, "Single and Multi-Domain Adaptive Allocation Algorithms for VNF Forwarding Graph Embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 98–112, 2019.

[15] B. Burns, J. Beda, K. Hightower, and L. Evenson, *Kubernetes: up and running*. " O'Reilly Media, Inc.", 2022.

[16] J. Turnbull, *Monitoring with Prometheus*. Turnbull Press, 2018.

[17] S. L. Ho and M. Xie, "The use of ARIMA models for reliability forecasting and analysis," *Computers & industrial engineering*, vol. 35, no. 1-2, pp. 213–216, 1998.

[18] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen *et al.*, "XGboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[19] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "A comparison of ARIMA and LSTM in forecasting time series," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 1394–1401.

[20] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," 2017.

[21] N. Makris, C. Zarafetas, S. Kechagias, T. Korakis, I. Seskar, and L. Tassiulas, "Enabling open access to LTE network components; the NITOS testbed paradigm," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, 2015, pp. 1–6.

Authorized licensed use limited to: University of Thessaly. Downloaded on December 08,2023 at 11:14:48 UTC from IEEE Xplore. Restrictions apply.