2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN) | 979-8-3315-2042-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICMLCN64995.2025.11140471

UNICORN: URLLC Network Traffic Classification and OOD Detection for O-RAN

Nasim Soltani*, Dante LoPriore†, Joshua Groen†, and Kaushik Chowdhury*

*Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX
†Electrical and Computer Engineering Department, Northeastern University, Boston, MA
nasim.soltani@utexas.edu, {lopriore.d, groen.j}@northeastern.edu, kaushik@utexas.edu

***Abstract*—The promise of Ultra-Reliable Low Latency Communication (URLLC) will transform verticals such as virtual reality, telesurgery, and tactile Internet among others. There are subtle differences in resource requirements between applications within URLLC category that may allow the network to perform fine-tuned resource allocation to satisfy stringent latency/jitter constraints. This paper proposes UNICORN, a neural network (NN)-based approach that aims to classify previously seen as well as detect new/emerging URLLC applications at the near real-time Radio access network Intelligence Controller (RIC), without any interactions from the application layer or from the user equipment (UE). The core approach leverages standardized key performance indicators (KPIs) exposed by an Open Radio Access Network (O-RAN) compliant cellular network stack. UNICORN is evaluated on a real-world dataset collected from a commercial cellular network using six URLLC smartphone applications with network KPIs obtained from a full-stack O-RAN implementation on the NSF Colosseum emulator. Results reveal classification accuracy of >96% with upto 88% true positive out-of-distribution (OOD) detection rate, and per class false positive rate as low as 8%.**

***Index Terms*—Open RAN, Traffic Classification, URLLC Application, OOD Detection, Key Performance Indicator.**

I. Introduction

Ultra-reliable low latency communication (URLLC) will usher in new interactive experiences for enhanced social interaction such as the tactile internet, revolutionize gaming and remote human experiences through virtual reality, as well as potentially save lives via telesurgery [1]. Each of these examples requires different latency and jitter thresholds, in turn requiring the network to provision resources differently for optimal outcomes [2]. Today there are numerous real-time applications such as voice calls, video chats, live streaming, and augmented reality (AR) that fall under the umbrella of URLLC, and share many common requirements, but are also sufficiently distinct from each other [3]. Thus, we believe that a *one-size-fits-all* approach is not sufficient for the network to generically support all URLLC applications, as there are considerable risks to over, or worse, under provision resources [4]. Consequently, 5G networks require precise traffic analysis to design fine-grained network slices optimized for Quality of Experience (QoE) control and service-specific functions, which can be aided by machine learning (ML) [5]. In fact, ML methods are shown to have superior performance over the non-ML methods for analyzing encrypted traffic [6].

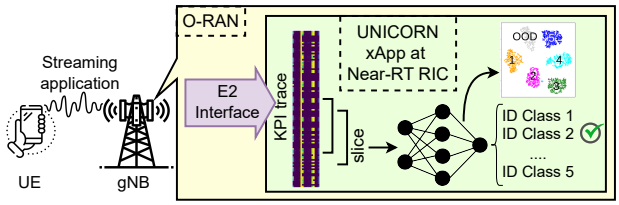

Fig. 1: The overview of UNICORN deployed as an xApp in the near-RT RIC, where an NN performs joint URLLC traffic classification and OOD detection.

• **Challenges.** There are, however, multiple challenges in deploying ML for traffic analysis: (i) To prevent numerous privacy concerns, the ML methods must not require access to user data. (ii) The ML methods can only be deployed in open and modular network systems, and their inputs are limited by the standardized interfaces that the network components provide. (iii) Despite the considerable recent interest in traffic analysis using ML in open networks, the previous work focus on classifying traffic to be a member of a fixed group. To the best of our knowledge there are no prior work on detecting new traffic classes that have not been seen during training.

• **Proposed Framework.** We propose UNICORN that enables the network to recognize which URLLC application is being used, while addressing the above challenges point-by-point: (i) UNICORN performs URLLC traffic analysis using only the Key Performance Indicators (KPIs) that are queried from the gNodeB (gNB) without accessing user data. (ii) It runs as an xApp in the near-Real-Time Radio access network Intelligence Controller (near-RT RIC) within Open Radio Access Network (O-RAN) systems [7], [8], and moreover, is able to operate on the data that the standard E2 interface provides (a.k.a., KPIs). (iii) It enables distinguishing new URLLC application classes from the ones that were seen during training. As shown in Fig. 1, UNICORN consists of a single light-weight neural network (NN) for joint in-library traffic classification and out-of-library (i.e., out-of-distribution (OOD)) traffic detection. UNICORN is evaluated using two different NN architectures trained and tested on a real-world URLLC dataset collected using a smartphone running 6 URLLC applications in different environments and different mobilities. The dataset is next replayed through the NSF Colosseum emulator [9] as an O-RAN digital twin, to achieve network KPI traces from

an emulated gNB, that are the NN data component in this paper. UNICORN achieves classification accuracy of >96% and OOD detection rates of upto 88% as true positive rate and as low as 8% as per class false positive rate.

Our contributions are as follows:

- We propose UNICORN as a unified pipeline for real-time URLLC in-library traffic classification and out-of-library detection that can be deployed as an xApp in the near-RT RIC within O-RAN. UNICORN finds the traffic class using network KPIs without accessing the user data.
- For detecting out-of-library URLLC traffic, we propose a novel non-parameteric OOD detection algorithm based on K-nearest neighbor (KNN) algorithm, that extracts features out of a hidden layer of the NN, characterizes in-distribution (ID) clusters, uses distances of test features from cluster centers, and relies on independent votes from K neighbors for ID/OOD decision.
- To evaluate UNICORN, we create a real-world dataset by collecting URLLC traces using a smartphone and generating network KPIs using Colosseum. The dataset is unique as it is the first dataset consisting of traffic traces collected during 42+ hours of *interactive operation* with 6 different URLLC applications in 3 different environments with different mobility scenarios. Another unique aspect is that the dataset comprises raw network traces collected from a smartphone that are not used in this paper, as well as network KPIs collected from Colosseum that are used in this paper as inputs to the NN model for URLLC application classification and OOD detection. We publicly share the URLLC dataset and Colosseum-generated KPIs [10], as well as the python source code [11] used to implement UNICORN.

II. Related Work

The work related to the scope of this paper can be classified into two main categories: work on traffic classification, and work that address discovering new classes.

Traffic Classification. Authors in [12] predict the slice classes of eMBB, mMTC, URLLC, and master slices to keep track of network load on each particular slice and allocate the less busy slices to the incoming traffic. Authors in [13] classify traffic to analyze slice congestion level and reallocate network slices. In the realm of traffic classification in O-RAN systems, where the KPIs are accessed instead of the user data-plane, authors in [7] propose a traffic analysis framework that consists of a traffic generator tool along with a simple CNN to classify slices of O-RAN traces into eMBB, mMTC, URLLC, and CTRL categories. Authors in [8] propose leveraging transformers for traffic classification and demonstrate they have better capabilities in learning temporal properties of O-RAN traces compared to simple CNNs. *To the best of our knowledge, there is no previous work on classifying traffic within the URLLC type in O-RAN.*

Discovering New Classes (i.e., OOD data). OOD detection is a well investigated topic in ML [14], [15]. In wireless communications, authors in [16] train a feature extractor with triplet loss and use KNN algorithm to detect OOD devices for RF fingerprinting, by comparing the distances of the test point from the nearest neighbors and the nearest neighbors from each other. Specifically in the realm of network traffic analysis, authors in [17] propose a feature-based method for OOD detection on mobile encrypted data, using a Long Short-Term Memory (LSTM) model for feature extraction. They obtain principal and residual principal components through Principal Component Analysis (PCA) and construct an OOD score to quantify deviation from the ID dataset. Authors in [18] propose a few-shot learning framework for traffic classification and OOD detection. They adopt the idea of Siamese networks, integrate it into the meta-learning framework, and use margin loss for detecting OOD data. *To the best of our knowledge, there is no previous work on detecting OOD traffic through processing network KPIs in O-RAN.*

III. Dataset Collection and KPI Generation

The procedure of collecting our dataset and generating KPIs has 3 steps: (i) 5G traffic capture, (ii) traffic emulation, and (iii) KPI capture, that are described in details in the following.

(i) 5G Traffic Capture. We capture 5G network traffic using a COTS Google Pixel 6a smartphone using PCAPdroid application, an open-source tool for capturing and analyzing network traffic on Android devices [19]. We collect URLLC traces from six different smartphone applications: Call of Duty, Twitch, Zoom, Microsoft Teams, Facebook, and Google Meet. We run the applications sequentially and record 5 to 10 minute traces per application in three different environments of indoor stationary, outdoor stationary, and outdoor walking. We gather 492 total traces over 42 hours of interaction with the six smartphone applications during several days, capturing different wireless channel conditions in each environment. The network traffic trace is initially saved on the smartphone in `.pcap` format. PCAPdroid provides a custom trailer that adds metadata associating an application label with each packet capture. Subsequently, the data is preprocessed to keep only the relevant meta-data (App name, packet number, time, source IP, destination IP, protocol, and packet length) and is saved as a `.csv` file.

(ii) Traffic Emulation. For traffic emulation, we use Colosseum [20] as an O-RAN digital twin [9] to replay the collected 5G traffic between an emulated User Equipment (UE) and a gNB. We use TRACTOR framework [7] to replicate the timing, length, and direction of all data sent between the UE and gNB, while anonymizing the payload within our experimental test bed. Furthermore, the O-RAN test bed simulates the channel conditions between the gNB and the UE based on measured conditions from a cellular system deployed in the real world. This setup enables us to accurately capture O-RAN KPIs and emulate the original communication in real time within our test bed.

(iii) KPI Capture. During replaying the traffic in the O-RAN test bed, we record all of the available KPIs and store them in a `.csv` file. Our O-RAN setup provides access

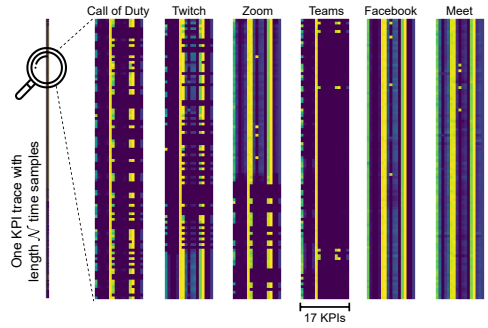

Fig. 2: Patterns within the 17 KPI traces for 6 different smartphone application classes in the dataset.

to 31 KPIs, as listed in [21], encompassing various low-level performance metrics. Before feeding these KPIs into the NN, we preprocess the data to eliminate KPIs with unique identifying information and administrative details, including slice assignments and scheduling policies, to further protect user privacy [7]. Furthermore, KPIs such as received signal strength indicator (RSSI) that are missing values in our Colosseum emulation are removed, thereby reducing the input dimensions without sacrificing essential information. The resulting dataset consists of 17 KPIs, including: *dl_mcs, dl_n_samples, dl_buffer (Bytes), tx_brate downlink, tx_pkts downlink, dl_cqi, ul_mcs, ul_n_samples, ul_buffer (Bytes), rx_brate uplink (Mbps), rx_pkts uplink, rx_errors uplink, ul_sinr, phr, sum_reqsted_prbs, sum_granted_prbs, and ul_turbo_iters*. The collected traces are normalized per-KPI by bringing all the values to the [0,1] interval and are fed to the NN for application classification.

The complete KPI dataset, which comprises 492 KPI traces equally distributed among six distinct application classes, amounts to 85 MB of data. Each trace is a rectangular matrix of $\mathcal{N}$ time samples collected from 17 KPIs, with characteristic patterns of each smartphone application. Fig. 2 displays sections of processed KPI traces as images with a single channel for six different applications. While some applications exhibit distinct KPI patterns, others display similarities, and there are instances where all applications show analogous patterns, which adds to the difficulty of classification and especially OOD detection in this dataset. The lengths of the traces vary, averaging to approximately 1500 time samples per trace.

IV. Classification and Out-of-Distribution Detection Pipeline

In this section, we describe the proposed method for joint URLLC application classification and OOD detection. We present a description of the NN architectures along with training and test processes in Section IV-A, and discuss the details of the novel OOD detection algorithm in Section IV-B.

A. Neural Network Architectures and Training/Test Processes

Neural Network Inputs and Outputs. The inputs to the NN are *slices* formed from 5G URLLC traces with 17 KPIs, as described in Section III. Each slice is a rectangle of 64 consecutive time samples from 17 KPIs structured as input with dimensions (1, 64, 17). We extract outputs from two different layers in the NN: (i) Feature vector with size 128 from one of the hidden layers, and (ii) Probability vector with size number of classes (# classes) from the last layer.

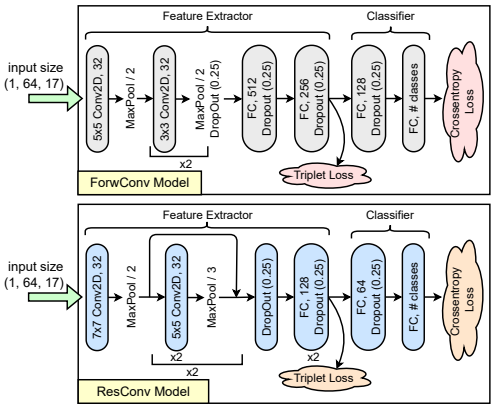

Fig. 3: ForwConv and ResConv NN architectures with 182k and 68k parameters, respectively.

Neural Network Architectures. We design two different custom NNs each implementing joint classification and OOD detection pipelines. As shown in Fig. 3, the first NN is a forward convolutional NN with 182k parameters denoted as ForwConv, and the second NN is a residual convolutional network with skip connections and 68k parameters denoted as ResConv. Each architecture has two main parts, the *feature extractor* and *classifier*, cascaded together. The feature extractor is mutual between the classification and OOD detection tasks, and consists of 2D convolutional, MaxPooling and dropout layers with dimensions specified in Fig. 3. The last layer on both feature extractors is a tensor of 128 activations. The classifier module is a series of fully connected (FC) layers with different sizes and dropout layers. The last layer of the classifier has size equal to the number of training set classes.

Training Process and Loss Functions. As shown in the top part of Fig. 4, NN training happens on a per-slice basis, where one feature vector with size 128 and one probability vector with size # classes are generated by the feature extractor and the classifier, respectively, for each input slice in the training batch. We define two different loss functions for training the NN as it is used for two different tasks of classification and OOD detection. As shown in Fig. 3, the first loss, $\mathcal{L}_1$, optimizes the output of the feature extractor. To perform a successful feature-based OOD detection, the feature vectors from the inputs of the same class need to fall as close together as possible in the feature space, and feature vectors from inputs of different classes need to fall as far from each other as possible. To satisfy this requirement, we use *triplet loss function* for $\mathcal{L}_1$. The second loss, $\mathcal{L}_2$, is calculated between the probability vector and the one hot representation of class label for the classification task, and is hence chosen as *categorical crossentropy loss function*. A total loss of $\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2$ is finally optimized to tune NN parameters through backward

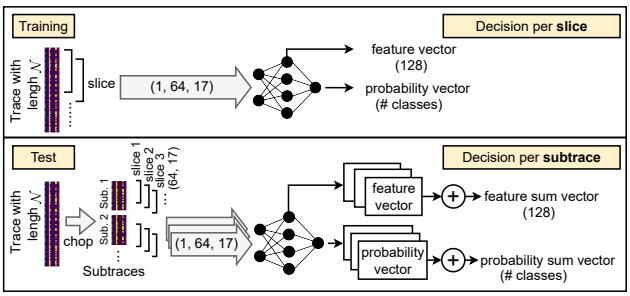

Fig. 4: Training and test processes.

propagation.

Test Process. During test, we chop each test trace with length $\mathcal{N}$ time samples to multiple non-overlapping *subtraces* of length 200, and then slice the subtrace with a stride of 1 to be fed to the NN. Each subtrace yields (200-64+1=)137 slices, and consequently, 137 feature vectors with size 128, and 137 probability vectors with size # classes, after the slices pass through the NN. We sum the feature vectors yielding from each subtrace to have a *feature sum vector* with size 128, and similarly, sum the probability vectors to have a *probability sum vector* with size # classes. To make a prediction for each subtrace we find the index of the largest value (i.e., $\arg\max$) in the probability sum vector. Furthermore, we use the feature sum vector to make ID/OOD decision for each subtrace. The test process is shown in the bottom part of Fig. 4.

B. Out-of-Distribution (OOD) Detection

Here, we describe the details of our novel OOD detection algorithm that examines the distances of the test feature from cluster center instead of distances from neighbors. Our proposed method also uses a different mechanism for taking votes from neighbors compared to the state-of-the-art [16]. Intuitively, the proposed OOD algorithm is not limited to any specific density or shape for embedding clusters, as opposed to measuring distance of test sample from neighbors [16] that relies on the assumption of dense population around cluster center and scattered points near the cluster edges. The proposed OOD detect algorithm spans through pre-deployment and post-deployment phases, as described in the following.

Pre-deployment: Characterizing ID Clusters. Characterizing ID clusters happens pre-deployment and after training, by passing the training set through the trained feature extractor, collect feature vectors, and calculate feature sum vectors (see Fig. 4) for different ID classes. In a training dataset containing J ID classes indexed with $j = 0, ..., J-1$, where each ID class has subtrace population of N_j, feature sum vectors of the encoder network are denoted as $y_j^{(n)}$ with $n = 0, ..., N_j - 1$. Each ID cluster needs to be characterized with a center, c_j, that is a vector of size I and a radius, r_j, that is a scalar. In this case, we gather all $y_j^{(n)}$ vectors belonging to each ID class j, and calculate a center, c_j, for each ID cluster using (1).

$$c_j = \frac{1}{N_j} \sum_{n=0}^{N_j-1} y_j^{(n)}, \qquad j = 0, 1, ..., J-1 \tag{1}$$

Algorithm 1: Characterizing In-Distribution (ID) Clusters

Inputs: Trained encoder network, Training set containing all ID classes
Set λ as 95%
Pass the training set through the trained encoder network and collect the ID features $y_j^{(n)}$s
center_list , radius_list = [ ] , [ ]
for all class j in ID classes **do**
  Calculate cluster center c_j using (1)
  center_list.append(c_j)
  for all $y_j^{(n)}$s indexed with n belonging to class j **do**
    distance_list = [ ]
    Calculate Euclidean distance of $y_j^{(n)}$ from its own cluster center c_j
    Append it as a scalar to distance_list
  end for
  Sort the distance_list in the ascending order
  Discard the last $1 - \lambda$ and pick the last element as r_j
  radius_list.append(r_j)
end for
Outputs: center_list, radius_list

Equation (1) calculates the I-dimensional mean of all I-dimensional feature sum vectors within each ID class. To compute the cluster radius, r_j, for each ID class indexed by j in the training set, we first determine the Euclidean distances of all feature sum vectors, $y_j^{(n)}$, from the cluster center, c_j, and sort them in ascending order. Next, we discard the bottom $(1 - \lambda)$ fraction of the distances, retaining only the top λ fraction. The discarded portion corresponds to feature sum vectors that are farther from the cluster center, c_j. From the retained portion, the largest value is selected as the cluster radius, r_j. By setting λ to a high value, such as 95%, we ensure that 95% of the ID feature sum vectors have distances to c_j smaller than or equal to the cluster radius r_j. In other words, 95% of the feature sum vectors for each ID class fall within their respective cluster. The steps for characterizing each ID cluster with a center and radius are summarized in Algorithm 1.

As the final step in the pre-deployment phase, we combine all the ID feature sum vectors from different ID classes into a set, and fit a KNN algorithm to them using the python API `NearestNeighbors()`.

Post-deployment: ID/OOD Decision Making. During the deployment phase, test subtraces from a mixture of ID and OOD classes are sliced and fed into the NN, their corresponding test features are generated by the feature extractor, and feature sum vectors are calculated. For each test feature sum vector y_{test} associated with each test subtrace, we find its K nearest neighbors among the ID features, as neighbor$_k$s. For each neighbor$_k$ belonging to the ID class j, we find its Euclidean distance from c_j, and denote it as d_k. We also calculate the Euclidean distance of y_{test} from c_j, and denote it as d_y. After this, we take a vote from each neighbor$_k$ on whether y_{test} belongs to an ID class or is OOD. We check two criteria for

Algorithm 2: Out-of-Distribution (OOD) Detection

Inputs: center_list, radius_list, test feature sum vector y_{test}
Return K nearest neighbors as neighbor_list = $\bigcup_{k=0}^{K-1}$ neighbor$_k$
vote_list = [ ]
for neighbor$_k$ in neighbor_list **do**
    Calculate d_k as Euclidean distance of neighbor$_k$ from its own cluster center c_j
    Calculate d_y as Euclidean distance of y_{test} from neighbor$_k$'s cluster center c_j
    v_k = ID **if** ($d_y \leq d_k$ **and** $d_y \leq r_j$) **else** OOD
    vote_list.append(v_k)
end for
v_{final} = ID **if** (ID in vote_list) **else** OOD
Output: v_{final}

the vote of neighbor$_k$ denoted as v_k, as in (2).

$$v_k = \begin{cases} \text{ID} & \text{if } d_y \leq d_k \text{and } d_y \leq r_j \\ \text{OOD} & \text{otherwise} \end{cases} \tag{2}$$

We derive a final vote for each y_{test} using the collective votes from its K neighbors as in (3).

$$v_{\text{final}} = \begin{cases} \text{ID} & \text{if any } v_k = \text{ID}, \quad k = 0, ..., K-1 \\ \text{OOD} & \text{otherwise} \end{cases} \tag{3}$$

Basically, we identify each test feature as OOD if none of its nearest neighbors vote it to be ID with respect to their own ID clusters. The steps to identify each test sample as ID or OOD are summarized in Algorithm 2.

V. Evaluations

We evaluate UNICORN on two sets of ID/OOD classes:

- **First set of classes:** Call of Duty, Facebook, Meet, Zoom, and Twitch as ID classes and Teams as OOD.
- **Second set of classes:** Call of Duty, Facebook, Meet, Zoom, and Teams as ID classes and Twitch as OOD.

For each set of classes, we create non-overlapping training, validation, and test partitions. We select 85% of the ID classes for training, ∼4% for validation, and ∼11% for test. We further expand the test set by adding traces of the OOD class to it, in a way that all ID and OOD classes in the test set have equal proportion of traces.

We train ForwConv and ResConv NNs shown in Fig. 3, on each set of ID/OOD classes, record training and validation losses, and plot them in Fig. 5. We observe that training loss is lower in ResConv, however, validation loss is closer to the training loss in ForwConv, which shows ResConv overfits to the training set while ForwConv has better generalization capabilities. After training, we feed the training set in each set of classes to each trained NN, extract feature vectors, calculate feature sum vectors, and characterize ID clusters using Algorithm 1. Then we test the trained models on a mutual test set consisting of ID and OOD classes with

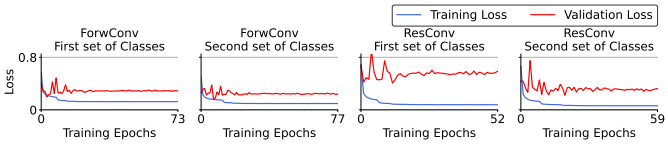

Fig. 5: Training and validation losses for two sets of classes, with ForwConv and ResConv NNs shown in Fig. 3.

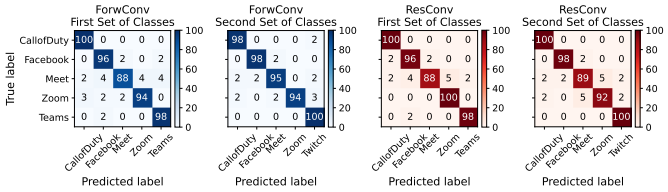

Fig. 6: Confusion matrices for two different sets of ID classes, with ForwConv and ResConv NNs shown in Fig. 3.

balanced trace populations, and extract probability vectors and feature vectors out of the last and second to last layers, respectively. We calculate probability sum and feature sum vectors as shown in Fig. 4.

Classification Accuracy. We obtain predicted labels by performing $\arg\max$ on probability sum vectors. We compare predicted labels with true labels to find the correct predictions, and define accuracy as the number of correct predictions, divided by the total number of predictions. We plot confusion matrices for ForwConv and ResConv trained with two different sets of ID classes in Fig. 6. We observe a near perfect classification accuracy of 96+% for all trained models.

OOD Detection Rate. To visualize the multi-dimensional clusters discussed in Section IV-B, we collect probability sum vectors (shown in Fig. 4) from different ID and OOD classes in the test set, and pass them through t-SNE [22] to reduce their dimensions to 2 and make them visualizable. We plot the 2D clusters in Fig. 7, for ForwConv and ResConv NNs and two different sets of ID/OOD classes. For the second set of classes with Twitch as OOD, we observe that ID and OOD clusters are well separated from each other for both NNs. However, with the first set of classes with Teams as OOD, we see that OOD test points are only fairly separate from the ID clusters. Moreover, after comparing ForwConv and ResConv plots, ForwConv (i.e., the larger NN) shows better cluster separation for both OOD classes, and consequently, we expect it to yield a larger OOD detection rate.

We make ID/OOD decision for each test subtrace by passing the corresponding probability sum vector through Algorithm 2. We define OOD detection rate for each class as the ratio of subtraces detected as OOD divided by the total number of subtraces for that class in the test set. Accordingly, we calculate OOD detection rate for all of the classes, regardless of being ID or OOD, in each of the experiments shown in Fig. 7, by setting nearest neighbor K as 5, 10, and 15, and plot them in Fig. 8. We observe that within each NN, Twitch as OOD is showing higher OOD detection rate compared to Teams as OOD, and within each set of classes, ForwConv

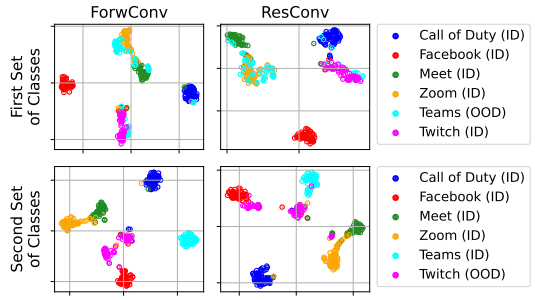

Fig. 7: 2D clusters for two different sets of ID/OOD classes and ForwConv and ResConv NNs shown in Fig. 3.

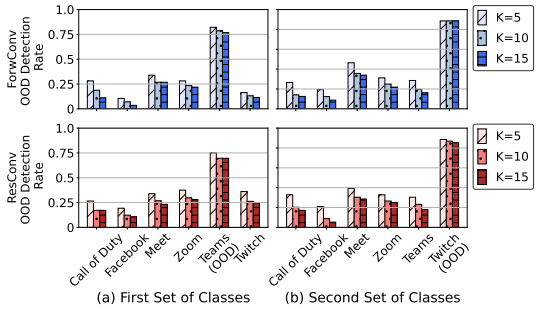

Fig. 8: OOD detection rate for (a) First set of classes and (b) Second set of classes for NNs shown in Fig. 3.

shows higher true positive OOD detection rate and overall lower false positives, which are both consistent with our observations in Fig. 7. We also observe that increasing K from 5 to 15 decreases both true and false positive rates, however, it has a greater impact on the false positive rate (i.e., OOD detection rate for ID classes). With ForwConv, and K=15, we observe true positive OOD detection rate of 88% for Twitch, and per class false positive rate as low as 8%.

VI. Conclusion

In this paper, we proposed UNICORN as a framework for joint URLLC network traffic classification and OOD detection within O-RAN systems. We described our real-world KPI trace dataset collected from a commercial cellular network using 6 smartphone applications with URLLC traffic type. We presented the details of the proposed NN-based classification and OOD detection methods that run as an xApp in the near-RT RIC. We evaluated UNICORN on two sets of ID/OOD classes, using two different custom NN architectures named as ForwConv and ResConv. We showed 96+% average accuracy in classifying 5 URLLC applications. Furthermore, we showed superior performance of ForwConv over ResConv in OOD detection by yielding upto 88% true positive OOD detection rate with false positive rate as low as 8%.

Acknowledgment

This work has been supported by NSF grants CNS 2229444 and CIRC 2120447.

References

[1] M. E. Haque, F. Tariq, M. R. Khandaker, K.-K. Wong, and Y. Zhang, "A survey of scheduling in 5g urllc and outlook for emerging 6g systems," *IEEE access*, vol. 11, pp. 34372–34396, 2023.

[2] R. Ali, Y. B. Zikria, A. K. Bashir, S. Garg, and H. S. Kim, "URLLC for 5G and Beyond: Requirements, Enabling Incumbent Technologies and Network Intelligence," *IEEE Access*, vol. 9, pp. 67064–67095, 2021.

[3] M. Alrabeiah, U. Demirhan, A. Hredzak, and A. Alkhateeb, "Vision aided URLL communications: Proactive service identification and co-existence," in *2020 54th Asilomar Conference on Signals, Systems, and Computers*, pp. 174–178, IEEE, 2020.

[4] Z. Li, M. A. Uusitalo, H. Shariatmadari, and B. Singh, "5G URLLC: Design challenges and system concepts," in *15th international symposium on wireless communication systems (ISWCS)*, pp. 1–6, IEEE, 2018.

[5] C. Gijón, M. Toril, M. Solera, S. Luna-Ramírez, and L. R. Jiménez, "Encrypted traffic classification based on unsupervised learning in cellular radio access networks," *IEEE Access*, vol. 8, pp. 167252–167263, 2020.

[6] T. T. Nguyen and G. Armitage, "A Survey of Techniques for Internet Traffic Classification using Machine Learning," *IEEE communications surveys & tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[7] J. Groen, M. Belgiovine, U. Demir, B. Kim, and K. Chowdhury, "Tractor: Traffic analysis and classification tool for open ran," in *ICC 2024 - IEEE International Conference on Communications*, 2024.

[8] M. Belgiovine, J. Gu, J. Groen, M. Sirera, U. Demir, and K. Chowdhury, "Megatron: Machine learning in 5g with analysis of traffic in open radio access networks," in *International Conference on Computing, Networking and Communications (ICNC)*, 2024.

[9] M. Polese, L. Bonati, S. D'Oro, P. Johari, D. Villa, S. Velumani, R. Gangula, M. Tsampazi, C. P. Robinson, G. Gemmi, *et al.*, "Colosseum: The Open RAN Digital Twin," *arXiv preprint arXiv:2404.17317*, 2024.

[10] Nasim Soltani, Dante LoPriore, "UNICORN Dataset." https://genesys-lab.org/unicorn.

[11] Nasim Soltani, "UNICORN Repository." https://github.com/nasimsoltani/unicorn.

[12] A. Thantharate, R. Paropkari, V. Walunj, and C. Beard, "DeepSlice: A deep learning approach towards an efficient and reliable network slicing in 5G networks," in *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pp. 0762–0767, IEEE, 2019.

[13] M. S. Abood, H. Wang, D. He, M. Fathy, S. A. Rashid, M. Alibakhshikenari, B. S. Virdee, S. Khan, G. Pau, I. Dayoub, *et al.*, "An LSTM-based network slicing classification future predictive framework for optimized resource allocation in C-V2X," *IEEE Access*, vol. 11, pp. 129300–129310, 2023.

[14] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized Out-of-Distribution Detection: A Survey," *arXiv preprint arXiv:2110.11334*, 2021.

[15] J. Zhang, J. Yang, P. Wang, H. Wang, Y. Lin, H. Zhang, Y. Sun, X. Du, K. Zhou, W. Zhang, *et al.*, "OpenOOD v1.5: Enhanced Benchmark for Out-of-Distribution Detection," *arXiv preprint arXiv:2306.09301*, 2023.

[16] G. Shen, J. Zhang, A. Marshall, and J. R. Cavallaro, "Towards Scalable and Channel-Robust Radio Frequency Fingerprint Identification for LoRa," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 774–787, 2022.

[17] Y. Tong, Y. Chen, G. B. Hwee, Q. Cao, S. G. Razul, and Z. Lin, "A Method for Out-of-Distribution Detection in Encrypted Mobile Traffic Classification," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2024.

[18] G. Miao, G. Wu, Z. Zhang, Y. Tong, and B. Lu, "SPN: A Method of Few-shot Traffic Classification with Out-Of-Distribution Detection Based on Siamese Prototypical Network," *IEEE Access*, 2023.

[19] PCAPdroid, "Open source PCAPdroid." https://github.com/emanuele-f/PCAPdroid.

[20] L. Bonati, P. Johari, M. Polese, S. D'Oro, S. Mohanti, M. Tehrani-Moayyed, D. Villa, S. Shrivastava, C. Tassie, K. Yoder, *et al.*, "Colosseum: Large-scale wireless experimentation through hardware-in-the-loop network emulation," in *2021 IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, pp. 105–113, IEEE, 2021.

[21] Joshua Groen, Mauro Belgiovine, "TRACTOR Repository." https://github.com/genesys-neu/TRACTOR.

[22] L. Van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of machine learning research*, vol. 9, no. 11, 2008.