

VERITAS: Verifying the Performance of AI-native Transceiver Actions in Base-Stations

Nasim Soltani*, Michael Löhning†, and Kaushik Chowdhury*

*Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX, USA

†National Instruments - Test and Measurement Group of Emerson, Dresden, Germany

nasim.soltani@utexas.edu, michael.loehning@emerson.com, kaushik@utexas.edu

Abstract—Artificial Intelligence (AI)-native receivers provide lower bit error rate (BER) compared to the traditional receiver, if they are deployed on the same data distribution as their training set. A major research problem is the uncertainty of whether a particularly trained AI-native receiver maintains its superior performance over the traditional receiver in different deployment environments. To this end, we propose VERITAS as a joint measurement-recovery post deployment framework for AI-native transceivers that continuously looks for distribution shifts in the received pilots and triggers finite re-training spurts. VERITAS leverages a novel out-of-distribution algorithm to detect potential changes in the channel profile, transmitter speed, and delay spread. As soon as such a change is detected, a traditional (reference) receiver is activated, which runs for a period of time in parallel to the AI-native receiver. Finally, VERITAS compares the bit probabilities of the AI-native and the reference receivers for the same received data inputs, and decides whether or not a retraining process needs to be initiated. Our evaluations reveal that VERITAS can detect changes in the channel profile, transmitter speed, and delay spread with 99%, 97%, and 78% accuracies, respectively, followed by timely initiation of retraining for 86%, 93.3%, and 94.8% of inputs in channel profile, transmitter speed, and delay spread test sets, respectively.

Index Terms—AI-native air interface, AI-native receiver, re-training, OOD detection, adaptive receiver, channel change, NN-based receiver, 5G receiver, DeepRx.

I. INTRODUCTION

Artificial intelligence native air interface (AI-AI) offers a fully AI-based interface for next-generation wireless communications, where AI is integrated in both data and control paths [1], [2]. AI-AI provides a myriad of flexibilities and opportunities for physical layer design, including but not limited to: merging data decoding and application in the physical layer, providing flexibility in the choice of waveform with respect to the radio hardware and environment constraints, obviating costly hardware implementation for each individual processing block by being fully AI-based, reduction in standardization need, and the possibility of physical and media access control (MAC) layer fusion [1], [2]. Furthermore, as data decoding and interpretation happens through neural networks (NNs) that have learned to map received data to originally transmitted bits, AI-based receivers previously showed to yield lower bit error rate (BER) compared to receivers with traditional signal processing blocks. Examples of such demonstration are shown with over-the-air data in [3], [4] and on real hardware in [5].

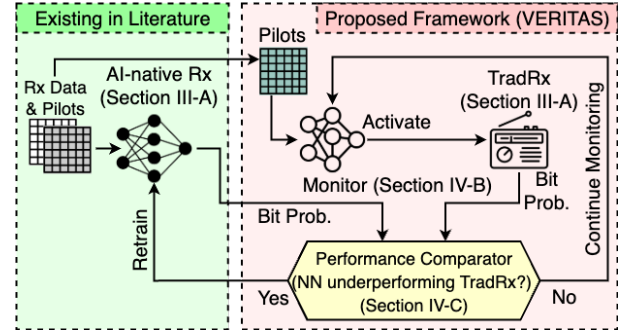


Fig. 1: VERITAS for verifying the performance of AI-native Receiver. The Monitor continuously scans the wireless channel, and as soon as it detects a change, it activates TradRx. Bit probabilities of AI-native receiver and TradRx are compared to find the underperforming receiver. If necessary, the AI-native receiver is updated through retraining to adapt to the new wireless channel.

Problem. Despite the several benefits that AI-AI provides for 6G communications, this newly proposed paradigm faces key challenges that need to be addressed before its successful deployment in 6G systems. For example, since wireless channel is a major contributor in the NN-based receiver performance, *verifying and maintaining the performance* of the AI-native receiver becomes a real challenge. NNs might not deliver the expected performance if they are deployed under wireless channels different from what they were trained in, as evident in several different examples: Recent work [6], [7] show that NN-based radio frequency (RF) fingerprinting accuracy drops drastically when training channels are different from deployment channels. Authors in [3] show retraining is necessary to maintain the performance of an NN-based channel estimator in new environments. Authors in [8] show that automatic modulation classification performance drops when the wireless channel changes, and propose transfer learning as an effective light retraining technique. The critical role of the wireless channel in AI-based wireless systems renders the performance verification and maintenance of AI-AI necessary.

Preliminaries. As shown on the left side of Fig. 1, we assume an NN-based receiver (a.k.a., AI-native receiver) that is responsible for converting received wireless signals (i.e., Rx Data & Pilots) to bits. Such models are well-explored in the literature in a number of prior works [9], [10], [11], [12], [13],

[14], [3], [4]. Due to practical limitations, the training set of the NN-based receiver cannot contain all possible data variations or signals recorded under all possible channels encountered in the real world. Instead, the NN-based receiver is trained on a number of channel profiles and mobility conditions. While it performs well under the seen configurations [15], [5], it is an open question whether it may suffer from a performance drop if deployed under a new wireless channel.

Limitation of Existing Solutions. To ensure maintained performance of the NN-based receiver, authors in [16] propose periodic retraining as a measure to adapt the receiver to new channel conditions. However, fixed-time-interval retraining imposes significant training computational complexity and requires dedicated computation resources. Furthermore, retraining an NN-based receiver in the field requires collecting signals that are *labeled* with transmitter-side bits under the new channel, which is not a trivial task. Therefore, unnecessary retraining must be avoided. Retraining must be applied only when we know that the performance of the NN-based receiver has definitely dropped compared to the traditional receiver, under the new channel conditions.

Proposed Solution. On the right side of Fig. 1, we propose VERITAS as a framework for verifying the performance of AI-native receiver in the field and limit its retraining to only necessary situations. VERITAS has 3 components: the Monitor, the traditional receiver (TradRx), and the Performance Comparator. The Monitor runs continuously in parallel to the AI-native receiver to observe the wireless channel and detect potential changes in the channel profile type, transmitter speed, or delay spread. As soon as such a change is detected, TradRx is activated and used as a comparison point against the AI-native receiver. The Performance Comparator compares output bit probabilities of the two receivers and determines the underperforming receiver. Notably, this step does not require the true bit labels or BER calculations. If the AI-native receiver is identified as the underperforming receiver, a retraining process is initiated to lightly retrain the AI-native receiver to ensure its maintained superior performance over the traditional receiver. The proposed Performance Comparator in VERITAS is able to operate on encoded as well as raw (i.e., uncoded) bits, which obviates the need for a costly decoding block within the proposed framework.

Contributions. Our contributions are as follows:

- We propose VERITAS as a framework for verifying the performance of an AI-native receiver, to ensure its maintained superior decoding performance compared to traditional receiver (Section IV-A).
- To demonstrate VERITAS works for generic AI-native receivers, we choose a widely used NN-based 5G receiver called DeepRx [15] as our AI-native receiver (Section III-A), which is designed to give lower BER compared to TradRx. We extensively analyze DeepRx performance for different training and test set configurations, and determine configurations where DeepRx yields higher BER compared to TradRx (Section III-B).
- We propose a wireless channel change detector called Monitor designed as a custom NN cascaded with a novel out-of-distribution (OOD) algorithm based on K-

nearest neighbor (KNN) [17]. The Monitor processes received 5G pilots and identifies any potential changes in the channel profile, transmitter speed, and delay spread. (Section IV-B).

- We propose an analytical method based on histogram binning to compare the output bit probabilities of the AI-native receiver against those of TradRx as reference. The proposed Performance Comparator compares output bit probabilities at the deployment phase and without having the true bit labels. This comparison determines if the AI-native receiver is underperforming with respect to the reference, which initiates a retraining process (Section IV-C).
- We pledge to publicly release our code [18] for VERITAS including pipelines for the Monitor and the Performance Comparator, upon the acceptance of this paper.

II. RELATED WORK

In this section, we summarize the closest related work in three different areas of AI-native receiver performance maintenance (Section II-A), wireless channel change detection (Section II-B), and OOD detection (Section II-C).

A. AI-native Receiver Performance Maintenance

The issue of performance drop in the NN-based receivers due to channel variations has been studied extensively. Authors in [16] propose a fixed time interval (periodic) retraining technique to adapt NN-based orthogonal frequency division multiplexing (OFDM) receivers to occasionally changing channel conditions. Naive periodic retraining is a way of maintaining performance of an NN-based receiver, however, it periodically imposes often unnecessary training computational complexity to the system as well as wastage of data frames that are used as the retraining dataset. Authors in [19] propose a denoising approach during training for learning OFDM channel coefficients. They construct their training set out of estimated channel coefficients of low noise signals, but dynamically add additive white Gaussian noise (AWGN) to inputs during training. This method makes the NN-based channel estimator robust to changes in the noise level, however, this does not solve the problem of transitioning between different wireless channels between training and deployment phases.

B. Wireless Channel Change Detection

Authors in [20] use the channel state information (CSI) of IEEE 802.11p signals for environment identification in V2V communication. They consider 5 different environments of rural line-of-sight (LOS), urban LOS, urban non-line-of-sight (NLOS), highway LOS, and highway NLOS for V2V communication and perform a multi-class classification using a deep convolutional NN, KNN, support vector machine, random forest, and Gaussian naive Bayes algorithms. They show superior performance of the NN over the other algorithms, however, they do not go beyond the fixed training set environments and do not show any method for identifying new environments. Authors in [21] classify user speeds in a 5G network using the reference signal received power (RSRP) passed through a deep NN. They categorize speeds between 0 and ∞ km/h

into 8 non-overlapping classes, with the last class spanning from 90 km/h to ∞ . This categorization encompasses all the possible speeds, however, known-class speed classification without OOD detection does not satisfy the requirements in our proposed AI-native receiver maintenance framework.

C. OOD Detection

Detecting OOD samples is a well-investigated problem in machine learning [22], [23]. In wireless communications, autoencoders have been vastly used for OOD detection. In such methods an autoencoder is trained to reconstruct an input, and the reconstruction error for known in-distribution (ID) inputs are averaged and recorded as a reference. At the deployment phase, all unknown inputs are fed to the autoencoder and their reconstruction errors are compared against the reference error. If the reconstruction error of the unknown test input is larger than the reference, the input is identified as an OOD input. Authors in [24] use a variational autoencoder and study the signal reconstruction error for identifying OOD modulation schemes. The disadvantage of autoencoder-based OOD detection is that completely different pipelines are needed for OOD detection and ID data classification. On the other hand, classification-based OOD detection methods provide a unified pipeline for both tasks. Authors in [25] detect unseen devices in the well-known RFMLS [26] WiFi and ADS-B datasets using a classification-based OOD detection method. They train their classifier NN with a custom loss function that has 3 components of intra-centroid loss, nearest neighbor loss and a final loss component that pushes the cluster centers away to spread in the space. However, their proposed method requires exposure of the NN to out-of-library devices (classes) that are not categorized into meaningful classes during training. Authors in [27] include a feature-based new device detection in their proposed LoRa RF fingerprinting scheme. They calculate the average of distances of test features from all of its K nearest neighbors, and compare it to a predefined λ value, and decide if the device is OOD or has been seen during training. This averaging process causes information loss and might degrade OOD detection performance. Furthermore, relying exclusively on distances from neighbors limits the methods to ID clusters that are dense in the center and scattered around the edges.

In the rest of this paper, we introduce a widely used NN-based receiver as our example AI-native receiver, and explore its performance for different training and test set configurations. Then, we describe and evaluate VERITAS as a framework for verifying the performance of this AI-native receiver to avoid its periodic and often unnecessary retraining.

III. PRELIMINARIES

In this section, we briefly describe different pipelines for data generation, the traditional receiver, and the NN-based receiver that we use in this paper as our example AI-native receiver (Section III-A). We explore the performance of the NN-based receiver in different training and test set configurations, and attempt to find cases where DeepRx shows a higher BER compared to TradRx and refer to them as *performance drop cases* (Section III-B).

A. Data Generation, TradRx, and AI-native Receiver Pipelines

Data Generation Pipeline. We generate data using Sionna libraries by synthesizing transmitter 5G radio frames containing random bits and passing them through 3GPP 38.901 tap-delay line channel models `tdl_a`, `tdl_b`, `tdl_c`, `tdl_d`, and `tdl_e`, that are implemented and available within Sionna. After the simulated channel, we also use the Sionna API `AWGN()` to add specific levels of noise to the data.

TradRx. Our traditional receiver that we refer to as TradRx, is based on least square (LS) channel estimation and linear minimum mean square error (LMMSE) equalization. To implement TradRx, different Sionna classes and functions including `OFDMDemodulator`, `LSChannelEstimator`, `LMMSEEqualizer`, and `Demapper` are used. The implemented TradRx is used as a reference for benchmarking the performance of AI-native receiver for different dataset configurations.

The 5G AI-native Receiver: DeepRx. As our AI-native receiver, we adopt a widely used fully convolutional 5G receiver with 672k parameters called DeepRx [15]. DeepRx interprets frequency domain I/Q samples in 5G subframes to their corresponding softbits (a.k.a., log likelihood ratios (LLRs)). The input of DeepRx is a (14, 72, 6) tensor that contains real and imaginary parts of the frequency domain received 5G subframes, raw estimated channel coefficients, and transmitter-side pilot symbols, stacked together in the last dimension. More details about DeepRx architecture can be found in [15]. We note that as the error correction block is not part of the DeepRx NN in [15], we do not include this block in the implementation of either DeepRx or TradRx. Therefore, without losing generality of our proposed method, all the BER results reported in the rest of this paper are reported for *uncoded* bits.

B. Exploration of DeepRx Performance Compared to TradRx

The superior BER performance of DeepRx compared to the traditional receiver is previously evaluated in the original paper by Nokia Bell Labs [15]. National Instruments also shows DeepRx BER comparisons against the traditional receiver in a real-time system prototype built using their universal software radio peripherals (USRPs) [5]. However, in both works the training and test datasets have the same configurations. Here, we attempt to identify corner cases where DeepRx BER increases above the traditional receiver BER (a.k.a., *performance drop cases*). While we limit our studies to DeepRx as a widely used 5G receiver, our core method can be deployed to any general AI-native receiver.

In our preliminary experiments, we vary three parameters—channel profile, transmitter speed, and delay spread—between the training and test sets, and study their impact on DeepRx BER. In all training and test datasets, data modulation scheme is set to 16QAM, and AWGN is added such that the ratio of energy per bit to the spectral noise density (E_b/N_0) is in range 0 to 20 dB with steps of 2 dB. Each training and test set contains 5000 and 500 uplink 5G radio frames, respectively, per E_b/N_0 level, and per combination of channel profile, transmitter speed, and delay spread. In each training run, the DeepRx model is fully trained for ~ 20 epochs. To measure

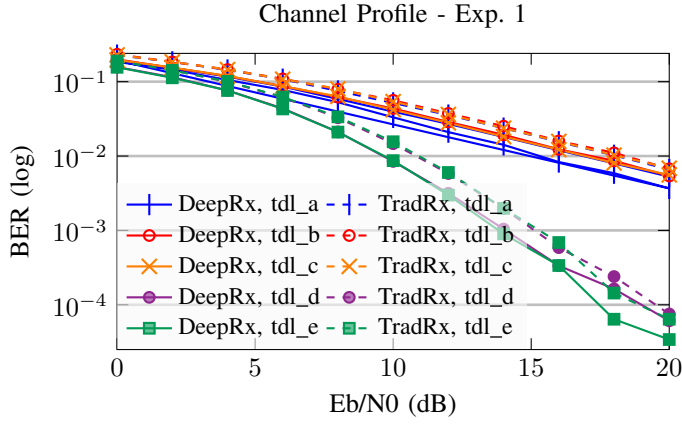


Fig. 2: BER vs. E_b/N_0 in Channel Profile - Exp. 1, where DeepRx is trained on tdl_a (NLOS) channel. During test, DeepRx outperforms TradRx in LOS and NLOS channel profiles, and therefore, no *performance drop case* is observed.

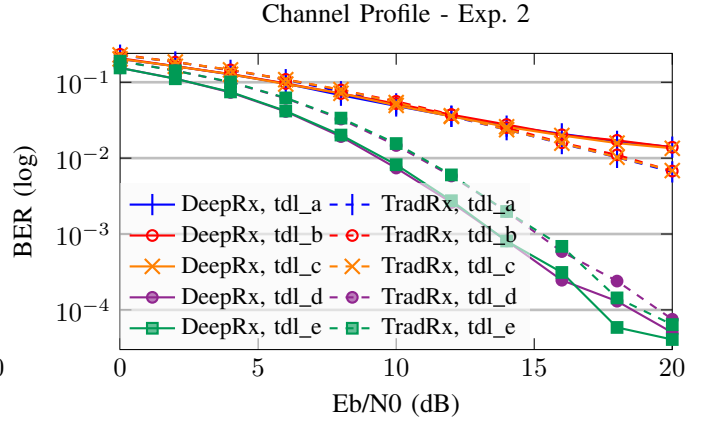


Fig. 3: BER vs. E_b/N_0 in Channel Profile - Exp. 2, where DeepRx is trained on tdl_d (LOS). During test, DeepRx underperforms TradRx for tdl_a, tdl_b, and tdl_c (i.e., NLOS) test channels, and therefore, a *performance drop case* is observed.

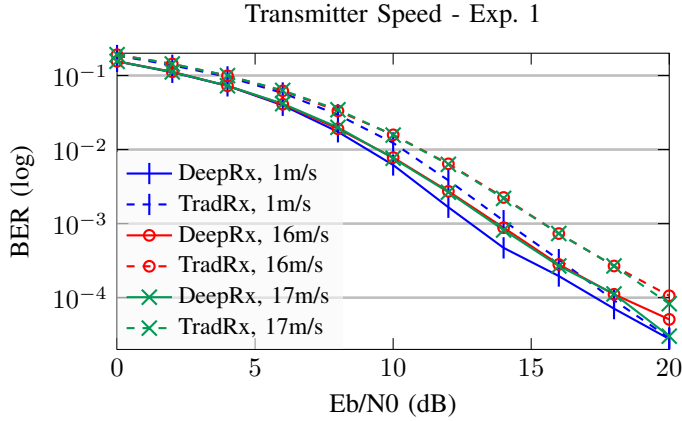


Fig. 4: BER vs. E_b/N_0 in Transmitter Speed - Exp. 1, where DeepRx is trained on higher speeds of 18, 19, and 20 m/s. During test, DeepRx outperforms TradRx and no *performance drop case* is observed.

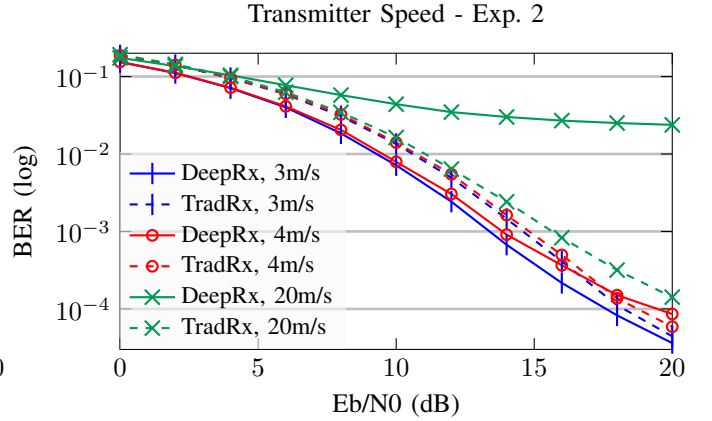


Fig. 5: BER vs. E_b/N_0 in Transmitter Speed - Exp. 2, where DeepRx is trained on lower speeds of 0, 1, and 2 m/s. During test, DeepRx underperforms TradRx in speeds 4 m/s and larger, which shows a *performance drop case*.

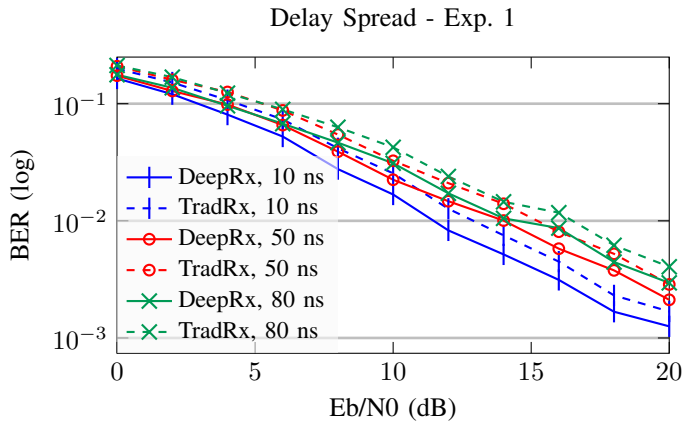


Fig. 6: BER vs. E_b/N_0 in Delay Spread - Exp. 1, where DeepRx is trained on higher delay spreads of 400, 450, and 500 ns. During test, DeepRx outperforms TradRx and no *performance drop case* is observed.

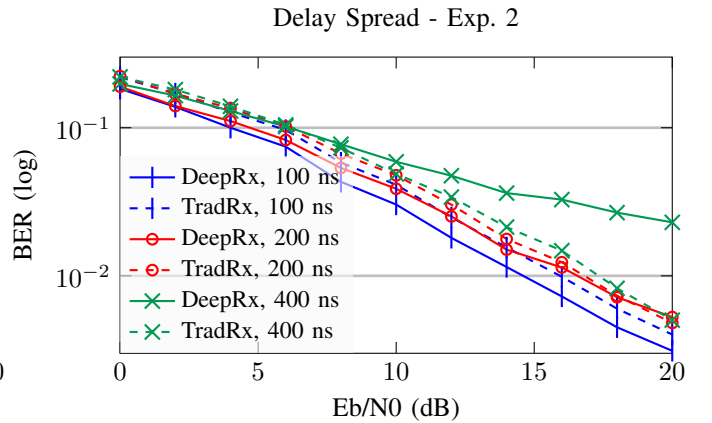


Fig. 7: BER vs. E_b/N_0 in Delay Spread - Exp. 2, where DeepRx is trained on lower delay spreads of 10, 50, and 80 ns. During test, DeepRx underperforms TradRx in 400 ns test set which shows a *performance drop case*.

#	Experiment name	BER vs. Eb/N0	Dataset	Channel Profile	Transmitter Speed (m/s)	Delay Spread (ns)
1	Channel Profile - Exp. 1	Fig. 2	Training	t dl_a	10	400
			Test	t dl_a, t dl_b, t dl_c, t dl_d, t dl_e	10	400
2	Channel Profile - Exp. 2	Fig. 3	Training	t dl_d	10	400
			Test	t dl_a, t dl_b, t dl_c, t dl_d, t dl_e	10	400
3	Transmitter Speed - Exp. 1	Fig. 4	Training	t dl_d	18, 19, 20	400
			Test	t dl_d	1, 16, 17	400
4	Transmitter Speed - Exp. 2	Fig. 5	Training	t dl_d	0, 1, 2	400
			Test	t dl_d	3, 4, 20	400
5	Delay Spread - Exp. 1	Fig. 6	Training	t dl_b	2	400, 450, 500
			Test	t dl_b	2	10, 50, 80
6	Delay Spread - Exp. 2	Fig. 7	Training	t dl_b	2	10, 50, 80
			Test	t dl_b	2	100, 200, 400

TABLE I: Summary of preliminary experiments description. Colored cells show the parameters that are variant between the training and test set datasets for each experiment.

		Channel Profile Exp. 2	Tx Speed Exp. 2	Delay Spread Exp. 2
Training	Channel Profile	t dl_d	t dl_d	t dl_b
	Tx Speed (m/s)	10	0, 1, 2	2
	Delay Spread (ns)	400	400	10, 50, 80
Test	Channel Profile	t dl_a, t dl_b, t dl_c	t dl_d	t dl_b
	Tx Speed (m/s)	10	≥ 4	2
	Delay Spread (ns)	400	400	> 200

TABLE II: Training and test configurations that lead to DeepRx underperforming TradRx.

TradRx BER and DeepRx BER in different Eb/N0 levels, the corresponding test set is passed through the TradRx, and the trained DeepRx model, respectively, and BER versus Eb/N0 is plotted in Figs. 2-7. It is expected that DeepRx BER is lower than TradRx BER in all Eb/N0 levels (i.e., DeepRx outperforms TradRx), otherwise that specific training/test configuration is flagged as a *performance drop case*. We perform 6 training experiments with different training and test set parameters as summarized in Table I.

Figures 2-7, show that DeepRx might provide higher BER compared to TradRx in three different cases: (i) change in the channel profile: if DeepRx is trained on a LOS channel such as t dl_d and deployed in NLOS channel profiles such as t dl_a, t dl_b, and t dl_c. (ii) change in the transmitter speed: if DeepRx is trained on a specific speed range such as speeds 0, 1, and 2 m/s and is tested on speeds that are higher than the training speeds by at least 2 m/s. (iii) change in the delay spread: if DeepRx is trained on low delay spreads such as 10, 50, and 80 ns and tested on higher delay spreads such as 400 ns. It should be noted that DeepRx performance drop cases are not limited to the above three configurations, however, these explored cases are summarized in Table II as example configurations that lead to DeepRx performance drop.

Next, we introduce VERITAS that automatically detects performance drop cases during the deployment of AI-native receiver.

IV. VERITAS FOR VERIFYING THE BER PERFORMANCE OF AI-NATIVE RECEIVERS

In this section, we describe VERITAS as a framework for verifying the performance of AI-native receiver. We describe

the overview of VERITAS and the interactions between its different components in Section IV-A. We provide the details of the *Monitor*, and the *Performance Comparator* in Sections IV-B and IV-C, respectively.

A. VERITAS System Overview

As shown in Fig. 1, VERITAS, is placed in parallel to the AI-native receiver and consists of three different components: the *Monitor*, the *Performance Comparator*, and the TradRx (introduced earlier in Section III-A). Among these components, the Monitor is the only component that is continuously active, while the other components are triggered based on certain conditions. The Monitor that is an NN cascaded with an OOD detection algorithm is pretrained on the same training set as the AI-native receiver. Therefore, the specific channel profiles, transmitter speeds, and delay spreads covered in the training set are considered ID data for the Monitor. The Monitor constantly observes the wireless channel by processing received pilots and flags potential OOD pilots as change in the wireless channel. Following this change detection, one could retrain the AI-native receiver to adapt it to the new wireless channel, however, not all changes might cause performance drop for the AI-native receiver (as observed in Figs. 2, 4, and 6), and retraining might be unnecessary. To avoid unnecessary retraining, at this point the Performance Comparator is activated and TradRx is triggered to run as the reference point in parallel to the AI-native receiver. The Performance Comparator compares bit probabilities generated by TradRx and the AI-native receiver, and determines the underperforming receiver. If the AI-native receiver outperforms TradRx, no retraining is required. In this case the TradRx and the Performance Comparator are deactivated and the Monitor continues observing the wireless channel to detect future potential changes. If the AI-native receiver underperforms TradRx, a retraining process is initiated to adapt the AI-native receiver to the new wireless channel. If a retraining process is initiated for the AI-native receiver, the Monitor needs to be retrained as well to update its set of ID classes to be able to continue detecting further changes in the wireless channel.

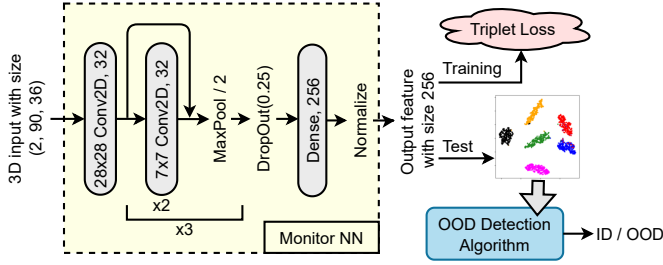


Fig. 8: The Monitor with an NN with $\sim 712k$ parameters that is trained with triplet loss function. During deployment the test features at the output of NN are fed to an OOD detection algorithm to give an ID or OOD decision for each input.

B. Wireless Channel Change Detector: Monitor

The job of the Monitor is to observe the wireless channel and detect potential changes in the channel profile, transmitter speed, or delay spread, which can be formulated as an OOD detection task. Among different varieties of OOD detection algorithms [23], [22], we adopt a feature-based OOD detection method from the post-hoc category, due to implementation simplicity and efficiency. We design a custom NN that generates vectorized features and train it on the training set containing the ID data classes, without getting exposed to any representation of OOD data during training. During deployment, the generated output features are fed to a novel distance-based OOD detection algorithm, to make an ID or OOD decision for each input. An overview of the Monitor is shown in Fig. 8.

In the following, we describe the input and output of the Monitor NN in Section IV-B1, the Monitor NN architecture along with the training and test processes in Section IV-B2, and the proposed OOD detection algorithm for detecting wireless channel changes in Section IV-B3.

1) *Input and Output Structure*: Here, we explain how Monitor input is prepared, and what output the Monitor provides.

Input. The wireless channel change detection happens through processing the frequency domain representation of the received 5G pilots. Specifically, we create a 3D matrix using pilots in 3 consecutive received frames. Constraining the input of the Monitor does not impose additional signal processing steps to the system, as the received frequency domain pilots are already prepared as an input component to DeepRx [15]. Depending on the selected pilot pattern, the number of pilot columns will be different which leads to different input sizes for the Monitor. In our selected pilot pattern (see the bottom of Fig. 9) each OFDMA subframe consists of complex-valued pilots with dimensions 36 and 3 along the frequency and time axes, respectively. We take pilot matrices from all the 10 subframes in 3 consecutive 5G radio frames and concatenate them along the time axis. We separate the real and imaginary parts of the pilots and form a matrix (tensor) with size (2, 90, 36) that is the input to the Monitor NN. The process of preparing inputs for the Monitor using 3 consecutive 5G radio frames each comprising 10 subframes is shown in Fig. 9.

Output. The output of the Monitor is a binary decision (ID or OOD) per input tensor.

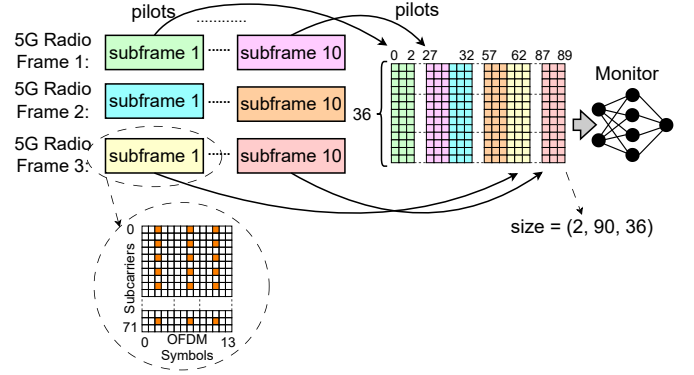


Fig. 9: To construct each input for the Monitor NN, received pilots in three 5G radio frames are concatenated to form 3D input tensors with dimensions (2, 90, 36).

2) *NN Architecture and Training Process*: For the Monitor NN architecture, we design a custom convolutional NN with residual blocks, consisting of convolutional, maxpooling, dense (or fully connected), and dropout layers, as shown in Fig. 8. We design the last layer of the NN to be a dense layer with output size $I = 256$, and normalize its output, x , as in (1), before sending it out of the NN.

$$y = \frac{x}{\max(|x|)} \quad (1)$$

Here, $\max(x)$ returns the maximum value among all the elements in vector x , and $|\cdot|$ is the absolute operator. We note that y is a vector of I elements, however, we do not denote its vector indexing in this paper for the sake of simplicity.

We train the Monitor NN with triplet loss function [28] to create clustered embeddings from input data. Triplet loss function operates on three input vectors: anchor feature and positive feature belonging to the same class, and negative feature belonging to a different class. Based on this, triplet loss function on a mini-batch comprising \mathcal{N} triples of input samples is defined as (2).

$$\mathcal{L} = \sum_{i=0}^{\mathcal{N}-1} \max\{\|x_a^{(i)} - x_p^{(i)}\|_2^2 - \|x_a^{(i)} - x_n^{(i)}\|_2^2 + \alpha, 0\} \quad (2)$$

In (2), $x_a^{(i)}$, $x_p^{(i)}$, and $x_n^{(i)}$ denote i^{th} anchor, positive, and negative features, respectively. $\|z_1 - z_2\|_2$ denotes Euclidean distance between any given two vectors z_1 and z_2 , and $\max(w_1, w_2)$ returns the largest value among w_1 and w_2 scalars. α is the triplet loss margin that we keep as the default value 1. In this way, the objective of training is to minimize \mathcal{L} that is a sum over \mathcal{N} loss components in each mini-batch.

The implication behind (2) is to map the positive features as close as possible to the anchor features, and to map the negative features as far as possible from the anchor feature in the feature space, and hence, to form distinct clusters for all the training set classes.

3) *OOD Detection Algorithm*: To detect a change in the wireless channel that is equivalent to identifying 5G pilot matrices as OOD, we propose a non-parametric OOD detection algorithm that processes the output features of the Monitor NN, y vectors. As the NN is trained with triplet loss function,

our algorithm relies on the assumption that features generated from inputs belonging to the same ID class fall in the same cluster, and features generated from inputs belonging to different ID classes form distinct clusters. The proposed algorithm flags each test feature as ID or OOD by comparing distances of the unknown test feature and known ID features to the center of the ID clusters. As distance calculations for the complete set of features within an ID cluster is computationally very costly, we limit our distance calculations to the K ID features that are closest to the test feature and are actually the most critical among all the ID features for ID or OOD decision. Based on this, we take advantage of the KNN [17] algorithm to find the K nearest neighbors of each test feature. In the following, we describe the processes of characterizing ID clusters and fitting a KNN model to them that happen pre-deployment and ID/OOD decision making for each test sample that happens at the deployment phase.

Characterizing ID Clusters: The multidimensional ID clusters are created by passing the ID classes (i.e., the training set) through the fully trained Monitor NN, in the pre-deployment phase. In a training dataset with J ID classes indexed with $j = 0, \dots, J - 1$, where each ID class has population N_j , output features of the Monitor NN are denoted as $y_j^{(n)}$ with $n = 0, \dots, N_j - 1$. 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 collect all output vectors, $y_j^{(n)}$, belonging to each ID class j , and calculate a center, c_j , for each ID cluster using (3).

$$c_j = \frac{1}{N_j} \sum_{n=0}^{N_j-1} y_j^{(n)}, \quad j = 0, 1, \dots, J - 1 \quad (3)$$

Equation (3) simply calculates an I -dimensional mean for all the I -dimensional features in each ID class. To calculate cluster radius, r_j , associated with each ID class j in the training set, we calculate the Euclidean distance of all training features, $y_j^{(n)}$, from c_j , and sort them in the ascending order. We discard the last $1 - \lambda$ portion of the sorted list and keep the first λ portion. The portion that is discarded is associated with those features that are far away from the cluster center, c_j . We pick the last (i.e., largest) value of the remaining list as the cluster radius, r_j . We set λ to a large value such as 95%, so that the distances of 95% of the ID features to the cluster center, c_j , are smaller than the cluster radius, r_j (i.e., 95% of ID features fall inside their respective clusters). The aforementioned steps for characterizing each cluster by a center and a radius are summarized in Algorithm 1.

Fitting the KNN Model. As the final step in the pre-deployment phase, we combine all the ID features from different ID classes in a single set and fit a KNN model to them, using `NearestNeighbors` class from `sklearn.neighbors` library.

Making ID/OOD Decision for Each Test Sample: At the deployment phase, the Monitor is tested on input samples that might belong to an ID class or might be OOD. For each output vector y_{test} generated using each test input, we find its K nearest neighbors, each denoted as neighbor_k with $k = 0, \dots, K - 1$. Obviously, each of these neighbors belong

Algorithm 1: Characterizing ID Clusters

- 1: **Inputs:** Trained Monitor NN, Training set containing all ID classes
 - 2: Set λ as 95%
 - 3: Pass the training set through the trained Monitor NN and collect the ID features, $y_j^{(n)}$ vectors
 - 4: center_list, radius_list = [], []
 - 5: **for** class j in ID_class_list **do**
 - 6: Calculate cluster center c_j using (3)
 - 7: center_list.append(c_j)
 - 8: distance_list = []
 - 9: **for** $y_j^{(n)}$ in class j **do**
 - 10: distance_list.append($\|y_j^{(n)} - c_j\|_2$)
 - 11: **end for**
 - 12: Sort the distance_list in ascending order
 - 13: distance_list = distance_list [start : $\lambda \times \text{end}$]
 - 14: $r_j = \text{distance_list}[\text{end}]$
 - 15: radius_list.append(r_j)
 - 16: **end for**
 - 17: **Outputs:** center_list, radius_list
-

Algorithm 2: OOD Detection

- 1: **Inputs:** center_list, radius_list, test feature y_{test}
 - 2: Return K nearest neighbors as n_list = $\bigcup_{k=0}^{K-1} \text{neighbor}_k$ and find their corresponding cluster centers using center_list and record them as c_list = $\bigcup_{k=0}^{K-1} c_j^{(k)}$
 - 3: **for** ($\text{neighbor}_k, c_j^{(k)}$) in zip (n_list, c_list) **do**
 - 4: $d_k = \|\text{neighbor}_k - c_j^{(k)}\|_2$, $d_y = \|y_{\text{test}} - c_j^{(k)}\|_2$
 - 5: $v_k = \text{ID}$ **if** ($d_y \leq d_k$ **and** $d_y \leq r_j$) **else** OOD
 - 6: **end for**
 - 7: $v_{\text{final}} = \text{ID}$ **if** ID \in vote_list **else** OOD
 - 8: **Output:** v_{final}
-

to one of the ID classes. For each neighbor_k belonging to the ID class j , we find the Euclidean distance of neighbor_k to its corresponding cluster center c_j , and denote it as d_k . We also calculate the Euclidean distance of y_{test} from neighbor_k 's associated cluster center, c_j , and denote it as d_y . After this, we take a vote from each neighbor_k belonging to ID class j , determining whether the test output feature y_{test} belongs to class j or not. We check 2 criteria to get the vote of neighbor_k denoted as v_k , as in (4).

$$v_k = \begin{cases} \text{ID} & \text{if } d_y \leq d_k \text{ and } d_y \leq r_j \\ \text{OOD} & \text{otherwise} \end{cases} \quad (4)$$

As shown in (4), a feature, y_{test} , is voted as ID by neighbor_k if the Euclidean distance of that feature to the cluster center of the neighbor is smaller or equal to the distance of the neighbor to its cluster center, and the distance of the feature from the cluster center of the neighbor is smaller than the cluster radius. Otherwise, the sample is voted as OOD by that neighbor.

We derive a final joint vote, v_{final} , for each y_{test} using the votes from its K neighbors as in (5).

$$v_{\text{final}} = \begin{cases} \text{ID} & \text{if any } v_k = \text{ID}, \quad k = 0, \dots, K-1 \\ \text{OOD} & \text{otherwise} \end{cases} \quad (5)$$

Basically, we identify each sample, y_{test} , 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 is summarized in Algorithm 2.

The highlights of the proposed OOD detection algorithm with respect to the state-of-the-art proposed in [29] and [27] are as follows:

- Our proposed OOD detection algorithm performs well in sparse clusters with lower density in the center and higher density around the edges, due to being dependent on the ID cluster center and radius instead of the distance of the test sample from its nearest ID neighbors.
- Our proposed algorithm utilizes triplet loss (Section IV-B2) instead of supervised contrastive loss [30] used in [29] during training that allows for good true OOD detection rate as well as low false positive rate.
- We use a custom NN architecture for OOD detection and show that non-parametric feature-based OOD detection is applicable to 5G wireless data besides the benchmark image datasets of CIFAR, SVHN, etc. demonstrated in [29].

If the Monitor detects a change in the wireless channel, the Performance Comparator is activated that is described next.

C. Performance Comparator

The job of the Performance Comparator in VERITAS is to decide if the AI-native receiver (e.g., DeepRx) needs to be retrained. It compares the bit probabilities generated by TradRx and DeepRx to determine the receiver with higher BER, and initiates a retraining process only if DeepRx yields higher BER. Obviously, this comparison happens using predicted bit probabilities without actual BER calculation or access to true bit labels. The Performance Comparator is able to operate on probabilities associated with encoded as well as uncoded bits, which obviates the need to include a costly decoding operation within VERITAS.

As soon as the Performance Comparator is activated by the Monitor, it triggers TradRx and runs it in parallel to DeepRx to decode the same received 5G radio frames for a specific time duration. We collect the softbits (i.e., LLRs) out of both receivers in this time duration, and convert them to bit probabilities. Bit probability P is calculated using its corresponding LLR through (6).

$$P = \frac{1}{1 + e^{\text{LLR}}} \quad (6)$$

In the hard decoding method, if P is less than or equal to 0.5 the bit is translated to logical ‘0’, and if P is greater than 0.5 the bit is translated to logical ‘1’. Since the same received 5G radio frames are passed through DeepRx and TradRx, ideally the same bit probabilities should be generated by both receivers. However, we discover that in practice this is not the case. We find bit probabilities of TradRx and DeepRx to be different for the same radio frames, and even more, we find

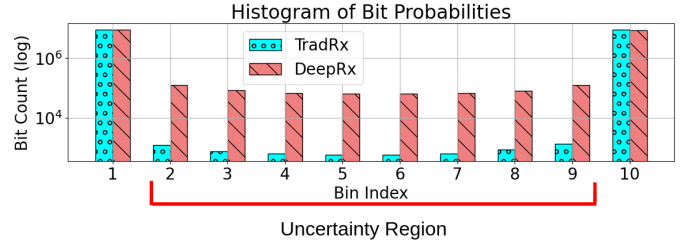


Fig. 10: Histogram of 4.5 million bit probabilities generated by passing the same test set through TradRx and DeepRx.

these probabilities to be related to each receiver’s BER. Based on our findings, we derive an empirical method that is inspired by histogram binning approach [31], which is a basic scheme for calibrating NN predicted probabilities. We note that we only leverage the “binning” concept without performing any sort of calibration or modification on DeepRx bit probabilities. We detail the proposed method in the following.

As each bit probability is a value between 0 and 1, we break the range 0 to 1 into 10 non-overlapping bins indexed with b , as in (7).

$$\text{bin}_b : \begin{cases} [\frac{b-1}{10}, \frac{b}{10}) & b = 1, 2, 3, \dots, 9 \\ [\frac{b-1}{10}, \frac{b}{10}] & b = 10 \end{cases} \quad (7)$$

Next, we categorize all the output bits from TradRx and DeepRx into these bins, based on their probability values, and count the number of bits in each bin. The histogram created for 4.5 million output bits of DeepRx and TradRx is shown in Fig. 10. In such a histogram, bin_1 and bin_{10} represent the most certain predictions for logical bits ‘0’ and ‘1’, respectively. On the other hand, bin_2 to bin_9 represent less certain predictions. We refer to these lower probability bins as the *uncertainty region*. We sum the bit counts in the uncertainty region and refer to them as $\mathcal{U}_{\text{DeepRx}}$ and $\mathcal{U}_{\text{TradRx}}$ for DeepRx and TradRx, respectively. The histogram bars in Fig. 10 are plotted for an example 5G radio frame dataset with transmitter speed set as 20 m/s and Eb/N0 as 20 dB (the same dataset studied in Experiment 4 in Table I). For this dataset, DeepRx provides a higher BER compared to TradRx, and accordingly, in Fig. 10, DeepRx shows higher bit counts in the uncertainty region compared to TradRx ($\mathcal{U}_{\text{DeepRx}} = 680\text{k}$ vs. $\mathcal{U}_{\text{TradRx}} = 6\text{k}$). Statistical analysis over the whole test set of 500 5G radio frames (i.e., 4.5 million bits) in different test speeds and different Eb/N0 levels, verifies the same relation between bit probabilities and bit errors: The receiver with the larger \mathcal{U} (i.e., taller histogram bars in the uncertainty region) yields higher BER. However, to comply with this observation, we require to run TradRx and DeepRx in parallel for 500 5G radio frames and collect 4.5 million bits, only to determine the underperforming receiver and potentially trigger retraining. This imposes significant elapsed time to VERITAS and might cause many cyclic redundancy check (CRC) fails before determining the underperforming receiver, which in turn reduces the overall communication throughput. The important question is *What is the smallest bit population that follows the observed rule regarding the relation of bit probabilities and receiver BER?* We evaluate the Performance Comparator and answer this question in Section V-B.

Algorithm 3: Performance Comparator

-
- 1: **Inputs:** $\text{LLR}_{\text{DeepRx}}, \text{LLR}_{\text{TradRx}}$
 - 2: Convert $\text{LLR}_{\text{DeepRx}}$ and $\text{LLR}_{\text{TradRx}}$ to P_{DeepRx} and P_{TradRx} , respectively, using (6)
 - 3: Create histogram bins for both P s using (7)
 - 4: Calculate $\mathcal{U}_{\text{DeepRx}}$ as sum of the bit count in bin_2 to bin_9 , for DeepRx outputs
 - 5: Calculate $\mathcal{U}_{\text{TradRx}}$ as sum of the bit count in uncertainty region (bin_2 to bin_9), for TradRx outputs
 - 6: Retraining = not needed **if** $\mathcal{U}_{\text{DeepRx}} \leq \mathcal{U}_{\text{TradRx}}$ **else** needed
 - 7: **Output:** Retraining
-

The described Performance Comparator algorithm is shown in Algorithm 3.

V. EVALUATIONS

In this section, we evaluate the performance of different components in VERITAS with respect to changes in the channel profile, the transmitter speed, and the delay spread. We evaluate the Monitor and the Performance Comparator in Sections V-A and V-B, respectively.

A. Wireless Channel Change Detector: Monitor

We train three Monitor NNs with triplet loss function, test the trained NNs and calculate and visualize the results in the following forms:

- (i) **2D Projection of Features.** We test each trained Monitor NN on all the ID and OOD data in the test set to get the 256-dimensional features. we use t-SNE [32] to reduce feature dimensions to 2, and plot them as scatter plots. We note that t-SNE is used only for visualization, and Algorithms 1 and 2 operate on the 256-dimensional features, without any dimension reduction.
- (ii) **OOD Detection Rate for Different K Values.** To numerically evaluate how separable the ID and OOD classes are, we pass the training set containing the ID classes through the trained Monitor NN and record the ID features. We characterize each 256-dimensional ID cluster by a 256-dimensional center c_j , and a scalar radius r_j through Algorithm 1. We run the OOD detection algorithm (i.e., Algorithm 2) with $\lambda = 0.95$ and nearest neighbor K values of 5, 10, and 15, on the unseen test set that contains ID and OOD classes. For each test class, each K , and each Eb/N0 level, we calculate OOD detection rate as the number of test feature vectors detected as OOD divided by the total number of test feature vector.
- (iii) **Sensitivity of Algorithm 2 to λ .** λ determines the radii r_j of ID clusters which are inputs to Algorithm 2. We study the sensitivity of Algorithm 2 to λ by measuring OOD detection rate while varying λ in range 0.5 to 1.0, with $K = 15$, at Eb/N0 levels 0, 10, and 20 dB.

We describe the experiments and results for detecting a change in the channel profile, transmitter speed, and delay spread in Sections V-A1, V-A2, and V-A3, respectively.

1) *Detecting a Change in the Channel Profile:* We evaluate the Monitor using dataset configurations guided by DeepRx performance shown in Fig. 3 and summarized in Table II. As we observe, in DeepRx *performance drop* happens if it is trained on LOS (i.e., tdl_d) and tested on NLOS (i.e., tdl_a,

tdl_b, and tdl_c). Therefore, one desired change detection is the transition between tdl_d channel profile to either one of the profiles tdl_a, tdl_b, or tdl_c. Based on this, we train the Monitor NN on the training dataset with configurations same as Experiment 2 in Table I that has only tdl_d channel profile. To be able to construct the triples of {anchor, positive, negative} for the triplet loss function, the training set requires to contain more than one class. The second class cannot be based on any of the unseen classes, however, should ideally still have a different distribution from the first class. Therefore, we artificially synthesize a second class derived from the tdl_d training samples and use it as auxiliary data to train the Monitor NN. To form the second class, we take the pilot matrix for each 5G radio frame for tdl_d class, and calculate the minimum and maximum values among the all the real and imaginary parts of pilots as u_{\min} and u_{\max} , respectively. Then, we create a new matrix with the same dimensions as tdl_d pilot matrix, and fill it with Uniform noise $\sim U(u_{\min}, u_{\max})$. We train the Monitor NN with triplet loss function to form distinct clusters for tdl_d and this auxiliary class.

Ideally the fully trained Monitor NN should be able to form distinct clusters for different ID classes and OOD data. The 2D projection of these 256-dimensional clusters is illustrated in Fig. 11 at Eb/N0 levels 0, 10, and 20 dB. We observe that the ID classes tdl_d (LOS) and Uniform noise form distinct clusters. Regarding the OOD classes, we see that features for all NLOS profiles (i.e., tdl_a, tdl_b, and tdl_c) fall in the same space but form clusters completely distinct from those of the ID classes. We observe that tdl_e (LOS) features completely overlap with the ID cluster tdl_d (LOS), since these two channel profiles are very similar, which is also evident in Fig. 3, where DeepRx model is trained on tdl_d (LOS), but faces no *performance drop* when it is tested on tdl_e (LOS). Based on this, the Monitor not being able to distinguish the OOD class tdl_e from the ID class tdl_d is not a problem, since DeepRx trained on tdl_d maintains its higher performance compared to TradRx, when tested on tdl_e.

The distinct ID and OOD clusters in all low, medium, and high Eb/N0 levels in Fig. 11(a), (b), and (c), respectively, can be justified by two reasons: First, the pattern of input to the Monitor that is the pseudo random sequence of 5G pilots with QPSK modulation scheme is a simple pattern, and it is not much affected by noise up to Eb/N0 = 0 dB. Second, the Monitor is trained on all Eb/N0 levels in range 0 to 20 dB with steps of 2 dB, and hence, good cluster distinction is observed in all Eb/N0 levels.

In Fig. 12, we observe that the OOD classes tdl_a, tdl_b, and tdl_c achieve 96%+ OOD detection rate in all Eb/N0 levels for $K = 5, 10, 15$. The average OOD detection rate for these NLOS channels over all Eb/N0 levels is 97%+ for all K values. We also observe that the ID classes tdl_d and Uniform noise achieve a low OOD detection rate (low false positive rate), which is desirable. This low rate reduces from averagely 9.3% to 5.2% as we increase K from 5 to 15. We observe that the OOD class tdl_e does not achieve a high OOD detection rate, however, transitioning from tdl_d in the training set to tdl_e during deployment does not cause performance drop for DeepRx as explained above.

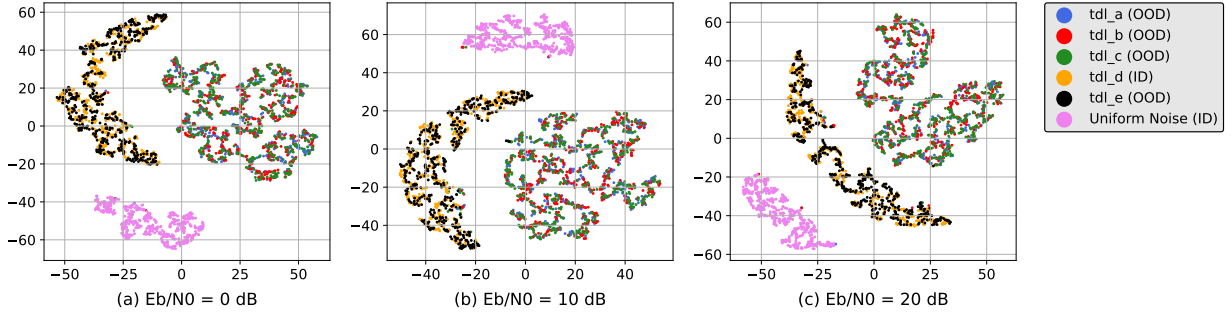


Fig. 11: 2D projection of feature vectors at the output of Monitor NN for detecting a change in the channel profile. The clusters are shown for different ID and OOD channel profile classes at E_b/N_0 levels (a) 0, (b) 10, and (c) 20 dB.

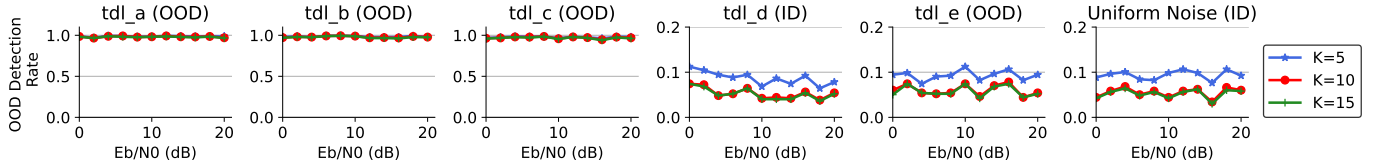


Fig. 12: OOD detection rate ($\lambda = 0.95$) for different test channel profiles with nearest neighbor parameter $K = 5, 10, 15$.

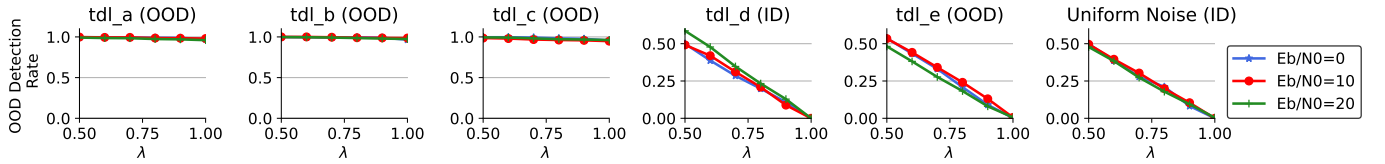


Fig. 13: Sensitivity of the proposed OOD detection algorithm (Algorithm 2) to λ for different test channel profiles with $K = 15$ in E_b/N_0 levels 0, 10, and 20 dB.

Fig. 13 shows OOD detection rate for different channel profile classes vs. λ at E_b/N_0 levels 0, 10, and 20 dB. We observe that varying λ does not impact OOD detection rate for true OOD classes (i.e., tdl_a, tdl_b, and tdl_c), however, for classes that are ID or similar to ID (i.e., tdl_d, tdl_e, and Uniform noise) the OOD detection decreases as λ increases.

2) *Detecting a Change in the Transmitter Speed:* Similar to Section V-A1, to define ID and OOD transmitter speed classes, we are guided by DeepRx performance shown in Fig. 5 and summarized in Table II. We train the Monitor NN with triplet loss function on the data with the same configurations as the training set of Experiment 4 in Table I, to form distinct clusters for transmitter speeds 0, 1, and 2 m/s.

We test the trained NN on the unseen test set that is a combination of different ID and OOD transmitter speed classes, and visualize 2D projection of their corresponding feature vectors at E_b/N_0 levels 0, 10 and 20 dB in Fig. 14. As observed, all the ID and OOD classes form distinct clusters, even in lower E_b/N_0 levels, as explained in Section V-A1.

Fig. 15 shows OOD detection rate for different ID and OOD transmitter speed classes for nearest neighbor K set as 5, 10, and 15. We observe low OOD detection rate for ID classes 0, 1, and 2 m/s across different E_b/N_0 levels, which is desirable as it shows low false positive rate. We see that increasing K from 5 to 15 reduces the average OOD detection rate from 10% to 3%, from 11% to 3%, and from 12% to 4% for ID classes 0, 1, and 2 m/s, respectively. For OOD classes 3, 4,

and 20 m/s we observe high OOD detection rate of 98%+ averaged over all E_b/N_0 levels for different K values, which is desirable as it shows high true positive rate.

Fig. 16 shows OOD detection rate vs. λ for different transmitter speed classes at E_b/N_0 levels 0, 10, and 20 dB. The trend is similar to Fig. 13 in except that OOD detection rate decreases for true OOD classes after a certain λ threshold. The implication is when cluster radius increases there is a higher chance that more OOD samples fall inside the cluster and are flagged as ID.

3) *Detecting a Change in the Delay Spread:* Similar to Sections V-A1 and V-A2 to form ID and OOD classes for Monitor NN, we are guided by DeepRx performance shown in Fig. 7 and summarized in Table II. We train the Monitor NN with triplet loss function on data with the same configurations as the training set of Experiment 6 in Table I, to form distinct clusters for ID delay spread classes 10, 50 and 80 ns.

After training, we test the trained NN on the unseen test set that is a combination of different ID and OOD delay spread classes, and visualize the 2D projection of the generated features for E_b/N_0 levels 0, 10, and 20 dB in Fig. 17.

We observe that with increasing the E_b/N_0 level beyond 0 dB, some ID clusters, specially 10 and 50 ns, tend to occupy a more independent space. However, overall the OOD samples are closer to ID clusters compared to Figs. 11 and 14 and are more difficult to distinguish even in the high E_b/N_0 levels. Because of this, we expect to see lower OOD detection rate

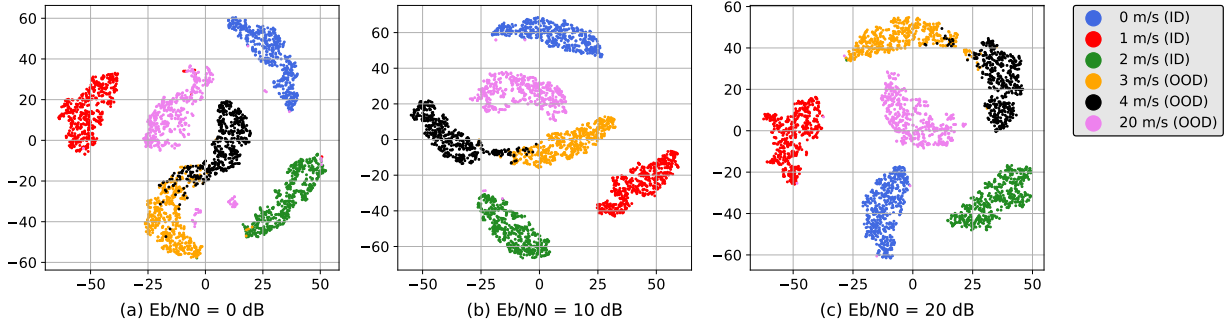


Fig. 14: 2D projection of feature vectors at the output of Monitor NN for detecting a change in the transmitter speed. The clusters are shown for different ID and OOD transmitter speed classes at E_b/N_0 levels (a) 0, (b) 10, and (c) 20 dB.

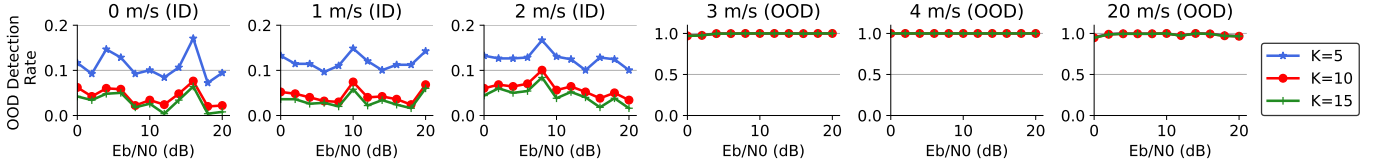


Fig. 15: OOD detection rate ($\lambda = 0.95$) for different test transmitter speeds with nearest neighbor parameter $K = 5, 10, 15$.

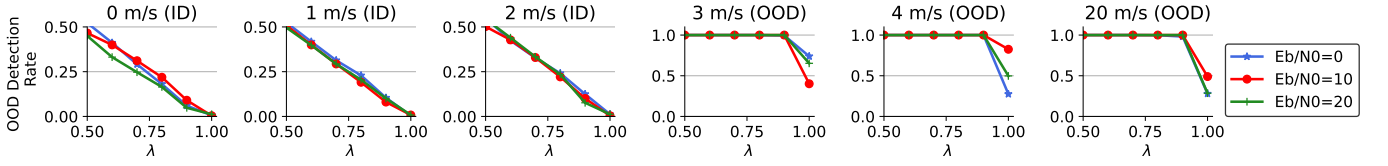


Fig. 16: Sensitivity of the proposed OOD detection algorithm (Algorithm 2) to λ for different test transmitter speeds with $K = 15$ in E_b/N_0 levels 0, 10, and 20 dB.

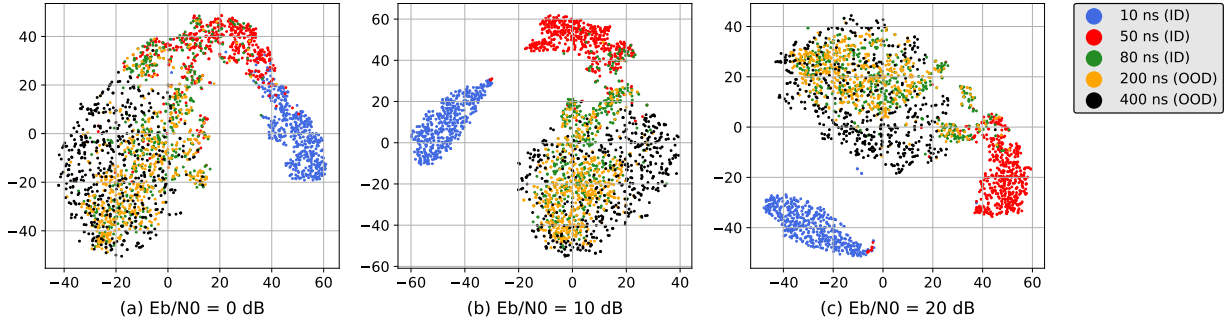


Fig. 17: 2D projection of feature vectors at the output of Monitor NN for detecting a change in the delay spread. The clusters are shown for different ID and OOD delay spread classes at E_b/N_0 levels (a) 0, (b) 10, and (c) 20 dB.

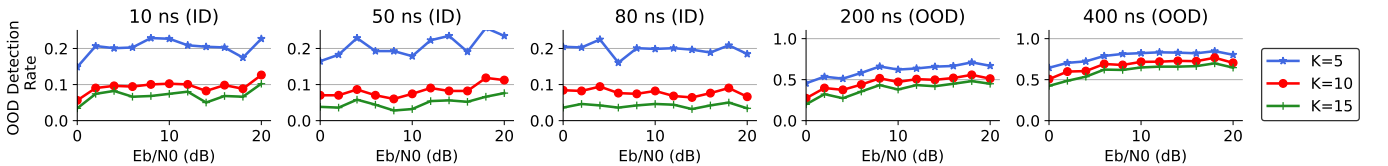


Fig. 18: OOD detection rate ($\lambda = 0.95$) for different test delay spread classes with nearest neighbor parameter $K = 5, 10, 15$.

for OOD classes compared to Sections V-A1 and V-A2.

Fig. 18 shows OOD detection rate for different ID and OOD delay spread classes for nearest neighbor K set as 5, 10, and 15. We observe low OOD detection rate for ID classes 10, 50, and 80 ns across different E_b/N_0 levels, which is desirable as

it shows low false positive rate. We see that increasing K from 5 to 15 reduces the average OOD detection rate from 20% to 6%, from 20% to 4%, and from 19% to 4% for ID classes 10, 50, and 80 ns, respectively. For the OOD classes 200 and 400 ns with $K = 5$, we observe an increase from 45% to 66%

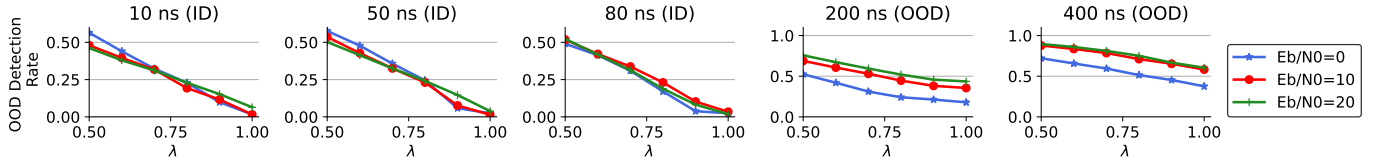


Fig. 19: Sensitivity of the proposed OOD detection algorithm (Algorithm 2) to λ for different test delay spread classes with $K = 15$ in E_b/N_0 levels 0, 10, and 20 dB.

and from 19% to 44%, respectively, as the E_b/N_0 increases from 0 to 20 dB. Furthermore, average OOD detection rate reduces from 61% to 38% and from 78% to 60%, for classes 200 and 400 ns, respectively, as K increases from 5 to 15. The relatively higher OOD detection rate for the OOD classes despite the partially overlapping clusters for 80, 200, and 400 ns in Fig. 17, shows that these clusters have some level of separation in the high-dimensional space.

Fig. 19 shows OOD detection rate vs. λ for different delay spread classes at E_b/N_0 levels 0, 10, and 20 dB. In all ID and OOD classes, we observe OOD detection rate decreases as λ (and consequently cluster radius) increases, which is expected for more closely located ID and OOD clusters.

Key takeaways.

- **OOD detection rate relation to 2D visualization.** It is expected that OOD classes that show visual separation from ID clusters in their 2D projection yield higher OOD detection rate, which is observed in Figs. 12 and 15.
- **Sensitivity to K .** If test feature vectors are closer to ID clusters, as K increases and more neighbors are queried, the chances that at least one neighbor votes for the test feature vector to be ID increases, and hence, the OOD detection rate decreases. This is consistent with t_{dl_d} , t_{dl_e} , and Uniform noise plots in Fig. 12, 0, 1, and 2 m/s plots in Fig. 15, and all plots in Fig. 18. However, for test feature vectors that show good separation from ID clusters, we expect that increasing K should not reduce the OOD detection rate, which is consistent with t_{dl_a} , t_{dl_b} , and t_{dl_c} plots in Fig. 12 and 3, 4, and 20 m/s plots in Fig. 15.
- **Sensitivity to λ .** In Figs. 13, 16, and 19, we observe that high λ values ensure low false positive rate for ID classes. For OOD clusters that are well-separated from ID clusters, larger λ values close to 1 yield high OOD detection rate. For OOD clusters that are in closer proximity to ID clusters or are partially overlapping with them, slightly lower λ values (e.g., 0.90-0.95) can provide an acceptable tradeoff between true and positive rates.

It is worth noting that the experiments conducted in Sections V-A1, V-A2, and V-A3 are designed to address the corner cases where the Monitor is imposed to as few as possible ID classes during training (i.e., 1-3), and tested on data where only one channel parameter changes (a.k.a., least amount of change that results in near OODs). Increasing the number of ID classes during training or varying multiple wireless channel parameters during test creates an easier OOD detection problem for the Monitor.

B. Performance Comparator

To evaluate the Performance Comparator we extract LLRs and calculate bit probabilities for DeepRx and TradRx, for all the test experiments of Channel Profile - Exp. 2, Speed - Exp. 2, and Delay Spread - Exp. 2 in Section III-B. As illustrated in Figs. 12, 15, and 18, for channel profile, transmitter speed, and delay spread, respectively, any of the inputs with ID or OOD true label might be predicted as OOD, even if it is with a low probability in the case of inputs with ID true labels. Therefore, the proposed Performance Comparator must be able to correctly trigger retraining for not only for OOD classes but also the ID classes. To evaluate this, we define an accuracy metric for the Performance Comparator based on the output from Algorithm 3. We evaluate the Performance Comparator on a *per-frame* basis, which means we collect the LLRs from DeepRx and TradRx for one 5G radio frame with 6 PBRs per subframe that is equivalent to 36k softbits from each receiver, and feed them to Algorithm 3. The 36k that is the number of softbits in each 5G radio frame is achieved through the following calculations:

$$\begin{aligned} & 10 \text{ [subframes]} \times 4 \text{ [16QAM modulation]} \times \\ & (72 \times 14 - 36 \times 3) \text{ [data minus pilots]} \\ & = 36000 \text{ [softbits]} \end{aligned}$$

The Performance Comparator determines whether or not DeepRx needs retraining once for every 5G radio frame. In Algorithm 3, we consider each prediction as a correct decision if the Performance Comparator flags retraining as “needed” and in fact $BER_{DeepRx} > BER_{TradRx}$ for the corresponding frame, or if it flags retraining as “not needed” and in fact $BER_{DeepRx} \leq BER_{TradRx}$ for that corresponding frame. For each test set, we calculate Performance Comparator accuracy as the number of correct decisions divided by the total number of decisions. We show Performance Comparator accuracy for different ID and OOD test sets of different channel profiles, transmitter speeds, and delay spreads in Sections V-B1, V-B2, and V-B3, respectively.

1) *Performance Comparator Accuracy in Different Channel Profiles:* In Fig. 20, where DeepRx is trained on t_{dl_d} channel profile, we observe Performance Comparator accuracies of 77%, 77%, 78%, 99%, and 99%, averaged over all E_b/N_0 levels, for test channel profiles t_{dl_a} , t_{dl_b} , t_{dl_c} , t_{dl_d} , and t_{dl_e} , respectively. This can be averaged to 86% accuracy for all the test channel profiles in all E_b/N_0 levels. Lowest accuracy in range 31-35% can be seen for the NLOS channel profiles t_{dl_a} , t_{dl_b} , and t_{dl_c} , in $E_b/N_0=14$ dB. Comparing this with Fig. 3 shows the low accuracy happens close to the E_b/N_0

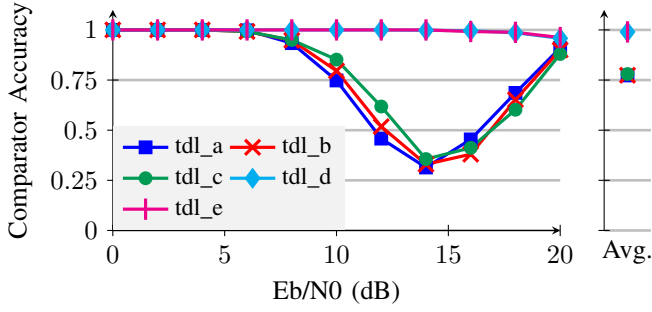


Fig. 20: Comparator accuracy when 5G radio frames with different channel profiles are passed through the TradRx and DeepRx trained on channel profile tdl_d. The plot on the right shows accuracy averaged over Eb/N0 levels.

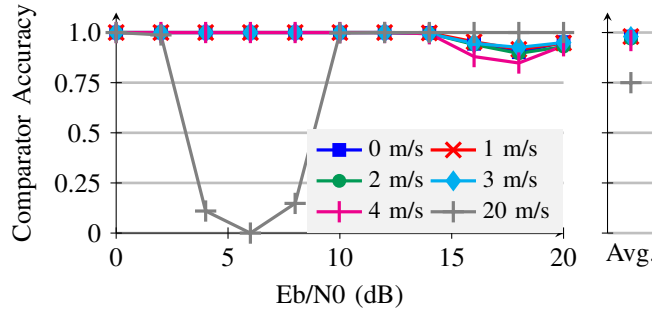


Fig. 21: Comparator accuracy when 5G radio frames with different transmitter speeds are passed through the TradRx and DeepRx trained on transmitter speeds 0, 1, and 2 m/s. The plot on the right shows accuracy averaged over Eb/N0 levels.

level where the two BER graphs of DeepRx and TradRx cross (i.e., 12 dB). This means the Performance Comparator makes incorrect decisions mostly when $\text{BER}_{\text{DeepRx}} \approx \text{BER}_{\text{TradRx}}$. At $\text{Eb}/\text{N}_0=14$ dB, $\text{BER}_{\text{DeepRx}}$ is only $2.6\text{e-}3$ higher than $\text{BER}_{\text{TradRx}}$. Therefore, an incorrect decision of “retraining not required” hurts the system BER a negligible amount of $2.6\text{e-}3$ higher BER, for $\sim 70\%$ of the frames.

2) *Performance Comparator Accuracy in Different Transmitter Speeds*: In Fig. 21, where DeepRx is trained on speeds 0, 1, and 2 m/s, we observe comparator accuracies of 98%, 98%, 97%, 98%, 96%, and 75%, averaged over all Eb/N0 levels, for test speeds 0, 1, 2, 3, 4, and 20 m/s, respectively. This can be averaged to 93.3% accuracy for all the test speeds in all Eb/N0 levels. Lowest accuracy of 0% can be seen for the highest speed of 20 m/s in $\text{Eb}/\text{N}_0=6$ dB. Comparing this with Fig. 5 shows the low accuracy happens close to the Eb/N0 level that the two BER graphs of DeepRx and TradRx cross (i.e., 4 dB). Similar to Section V-B1, the comparator makes incorrect decisions mostly when $\text{BER}_{\text{DeepRx}} \approx \text{BER}_{\text{TradRx}}$. At $\text{Eb}/\text{N}_0=6$ dB for speed 20 m/s, $\text{BER}_{\text{DeepRx}}$ is only $1.4\text{e-}2$ higher than $\text{BER}_{\text{TradRx}}$. Therefore, an incorrect decision of “retraining not required” hurts the system BER by only $1.4\text{e-}2$ higher BER.

3) *Performance Comparator Accuracy in Different Delay Spreads*: In Fig. 22, where DeepRx is trained on delay spreads 10, 50, and 80 ns, we observe comparator accuracies of 99%, 99%, 99%, 95%, and 82%, averaged over all Eb/N0 levels, for

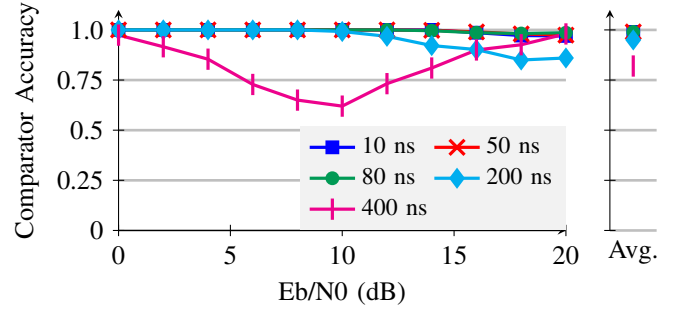


Fig. 22: Comparator accuracy when 5G radio frames with different delay spreads are passed through the TradRx and DeepRx trained on delay spreads 10, 50, and 80 ns. The plot on the right shows accuracy averaged over Eb/N0 levels.

test delay spreads 10, 50, 80, 200, and 400 ns, respectively. This can be averaged to 94.8% accuracy for all test delay spreads in all Eb/N0 levels. Lowest accuracy of 62% can be seen for the highest delay spread of 400 ns in $\text{Eb}/\text{N}_0=10$ dB. Comparing this with Fig. 7 shows the low accuracy happens close to the Eb/N0 level where the two BER graphs of DeepRx and TradRx cross (i.e., 6 dB). Similar to Sections V-B1 and V-B2, the comparator makes incorrect decisions mostly when $\text{BER}_{\text{DeepRx}} \approx \text{BER}_{\text{TradRx}}$. At $\text{Eb}/\text{N}_0=10$ dB for delay spread 400 ns, $\text{BER}_{\text{DeepRx}}$ is only $1.0\text{e-}2$ higher than $\text{BER}_{\text{TradRx}}$. Therefore, an incorrect decision of “retraining not required” hurts the system BER by only $1.0\text{e-}2$ higher BER for only $\sim 38\%$ of the 5G radio frames.

VI. DISCUSSION

In this section, we study the efficacy of VERITAS in terms of the computational overhead that it imposes to the AI-native receiver system, and we compare it against naive periodic retraining of the AI-native receiver.

The only component in VERITAS that is always active and continuously runs is the Monitor. The Monitor NN and the OOD detection algorithm have runtimes of $537 \mu\text{s}$ on Nvidia RTX 6000 GPU and $592 \mu\text{s}$ on CPU, respectively, per input. The total runtime for the Monitor entity adds up to averagely ~ 1.13 ms that is shorter than the duration of three radio frames (i.e., 30 ms) that construct the Monitor input, by a large margin. This shows that the Monitor can operate on streaming 5G frames, even if implemented in software. To avoid being limited to hardware and implementation-dependent runtime metrics, we focus on algorithmic level complexity for comparison and report the number of floating point operations (FLOPs) in the rest of this section.

We use Python package `ptflops` to count inference FLOPs for the Monitor NN as a PyTorch model. We also analyze different steps and equations that construct the OOD detection algorithm, the Performance Comparator, and the TradRx, and calculate FLOPs for each component in the VERITAS system as shown in Table IIIa.

Furthermore, we use Python package `keras_flops` to calculate FLOPs for training the AI-native receiver (i.e., DeepRx) using LAMB optimizer [33]. We achieve FLOP count of 4.003 G for training DeepRx on one 5G radio frame.

VERITAS Components	Inference GFLOPs	DeepRx	GFLOPs
Monitor NN	1.181	Inference on one radio frame	1.330
OOD Detection Algorithm (K=15)	0.012	Optimizer	0.013
Monitor Total	1.193	Training on one radio frame	4.003
Comparator	0.005	Training on 300 radio frames for 5 epochs	6004.5
TradRx	0.004		
VERITAS Total	1.202		

(a)

(b)

TABLE III: (a) The number of FLOPs for different components in VERITAS, computed for one input with size (2, 90, 30) corresponding to three 5G frames as explained in Fig. 8. (b) DeepRx FLOPs in different modes.

We consider the case where a change in the channel conditions occurs and DeepRx needs to be retrained. As we show in Section III training DeepRx from scratch requires ~ 5000 radio frames per Eb/N0, channel profile, transmitter speed, and delay spread, and continues for 20 epochs. We assume a minimum of ~ 300 radio frames and a minimum of 5 epochs required to retrain DeepRx in the field and fine-tune it on the new channel conditions. In this case, the total FLOPs for retraining DeepRx is estimated as $300 \times 4.003 \times 10^9 \times 5 = 6004.5$ GFLOPs, as shown in Table IIIb.

VERITAS aims to replace naive periodic retraining of the AI-native receiver to avoid unnecessary retraining, as discussed in Sections I and II-A. In periodic retraining, the frequency of retraining is configurable by the designer, and is upper bounded by the number of radio frames needed to retrain the AI-native receiver. In our AI-native example (i.e., DeepRx) in extreme cases and hypothetical setting, retraining can happen as frequent as every 300 radio frames (i.e., every 3 seconds) as discussed above.

According to Table III, the FLOPs count for retraining DeepRx matches the FLOPs of the Monitor running ($6004.5/1.193 \approx 5000$) times. This is equivalent to the Monitor processing ($5000 \times 3 \approx 15000$) radio frames that have a total time duration of 150 seconds. VERITAS does not provide lower computational complexity compared to periodic retraining, if DeepRx is scheduled to retrain less frequently than every ~ 150 seconds. However, VERITAS provides lower computational complexity, if DeepRx periodic retraining happens more frequently than every ~ 150 seconds. For example, assuming that periodic retraining of DeepRx is scheduled for every 3 seconds (a.k.a., the most frequent retraining possible), the Monitor has to run 100 times to process 300 frames. In this case, the total FLOPs for running the Monitor add up to ($100 \times 1.193 = 119.3$) GFLOPs, which is only ($119.3/6004.5 \approx 2\%$) the computational complexity of retraining DeepRx.

It should be emphasized that retraining of AI-native receiver requires not only dedicated compute resources, but also signals collected under the current channel with known transmit bit labels, whose collection is a significant communication overhead. VERITAS helps reduce both training computation and communication overhead while preventing AI-native receiver performance degradation caused by environment variations.

VII. CONCLUSION

In this paper, we proposed VERITAS as a framework for verifying the performance of AI-native receivers. VERITAS consists of a Monitor, a Performance Comparator, and a traditional receiver as the reference point. The Monitor that is an OOD detector NN constantly observes the wireless channel and detects changes in different parameters: channel profile (i.e., LOS or NLOS environment), transmitter speed, and delay spread. The proposed Monitor shows 99%, 97%, and 69% true OOD detection rate for channel profile, transmitter speed, and delay spread, respectively. As soon as a change in the wireless channel is detected, the Monitor activates a TradRx to be used as a reference receiver that runs in parallel to the NN-based receiver. The Performance Comparator compares the bit probabilities yielding from the same data inputs passing through DeepRx and TradRx and identifies the receiver with higher BER, to determine whether or not a retraining process needs to be started. The proposed Performance Comparator correctly triggers retraining with an average accuracy of 86%, 93.3%, and 94.8% for all channel profile, transmitter speed, and delay spread test sets, averaged over all Eb/N0 levels. VERITAS can assist in verifying and maintaining the BER performance of AI-native receivers in real-world deployments, such as the real-time AI-native receiver prototype integrated into NI's USRP-based research platform described in [5].

REFERENCES

- [1] J. Hoydis, F. A. Aoudia, A. Valcarce, and H. Viswanathan, "Toward a 6G AI-Native Air Interface," *IEEE Communications Magazine*, vol. 59, no. 5, pp. 76–81, 2021.
- [2] Ericsson, "Defining AI native: A key enabler for advanced intelligent telecom networks." <https://www.ericsson.com/en/reports-and-papers/white-papers/ai-native>.
- [3] N. Soltani, H. Cheng, M. Belgiovine, Y. Li, H. Li, B. Azari, S. D'Oro, T. Imbiriba, T. Melodia, P. Closas, *et al.*, "Neural Network-Based OFDM Receiver for Resource Constrained IoT Devices," *IEEE Internet of Things Magazine*, vol. 5, no. 3, pp. 158–164, 2022.
- [4] B. Azari, H. Cheng, N. Soltani, H. Li, Y. Li, M. Belgiovine, T. Imbiriba, S. D'Oro, T. Melodia, Y. Wang, *et al.*, "Automated Deep Learning-based Wide-band Receiver," *Computer Networks*, vol. 218, p. 109367, 2022.
- [5] National Instruments, "Prototyping a Real-Time Neural Receiver with USRP and OpenAirInterface." <https://www.ni.com/en/solutions/electronics/5g-6g-wireless-research-prototyping/prototyping-real-time-neural-receiver-usrp-openairint.html>.
- [6] A. Al-Shawabka, F. Restuccia, S. D'Oro, T. Jian, B. C. Rendon, N. Soltani, J. Dy, S. Ioannidis, K. Chowdhury, and T. Melodia, "Exposing the Fingerprint: Dissecting the Impact of the Wireless Channel on Radio Fingerprinting," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 646–655, IEEE, 2020.
- [7] N. Soltani, K. Sankhe, J. Dy, S. Ioannidis, and K. Chowdhury, "More Is Better: Data Augmentation for Channel-Resilient RF Fingerprinting," *IEEE Communications Magazine*, vol. 58, no. 10, pp. 66–72, 2020.
- [8] F. Meng, P. Chen, L. Wu, and X. Wang, "Automatic Modulation Classification: A Deep Learning Enabled Approach," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 11, pp. 10760–10772, 2018.
- [9] P. Jiang, T. Wang, B. Han, X. Gao, J. Zhang, C.-K. Wen, S. Jin, and G. Y. Li, "AI-aided OFDM Receiver: Design and Experimental Results," *arXiv preprint arXiv:1812.06638*, 2018.
- [10] F. A. Aoudia and J. Hoydis, "End-to-end Learning for OFDM: From Neural Receivers to Pilotless Communication," *IEEE Transactions on Wireless Communications*, 2021.
- [11] J. Zhang, C.-K. Wen, S. Jin, and G. Y. Li, "Artificial Intelligence-aided Receiver for a CP-free OFDM System: Design, Simulation, and Experimental Test," *IEEE Access*, vol. 7, pp. 58901–58914, 2019.
- [12] Z. Zhao, M. C. Vuran, F. Guo, and S. Scott, "Deep-waveform: A Learned OFDM Receiver Based on Deep Complex Convolutional Networks," *arXiv preprint arXiv:1810.07181*, 2018.

- [13] H. Ye, G. Y. Li, and B.-H. Juang, "Power of Deep Learning for Channel Estimation and Signal Detection in OFDM Systems," *IEEE Wireless Communications Letters*, vol. 7, no. 1, pp. 114–117, 2017.
- [14] X. Gao, S. Jin, C.-K. Wen, and G. Y. Li, "ComNet: Combination of Deep Learning and Expert Knowledge in OFDM Receivers," *IEEE Communications Letters*, vol. 22, no. 12, pp. 2627–2630, 2018.
- [15] M. Honkala, D. Korpi, and J. M. Huttunen, "Deeprx: Fully convolutional deep learning receiver," *IEEE Transactions on Wireless Communications*, vol. 20, no. 6, pp. 3925–3940, 2021.
- [16] M. B. Fischer, S. Dörner, F. Krieg, S. Cammerer, and S. ten Brink, "Adaptive NN-based OFDM Receivers: Computational Complexity vs. Achievable Performance," in *2022 56th Asilomar Conference on Signals, Systems, and Computers*, pp. 194–199, IEEE, 2022.
- [17] L. E. Peterson, "K-Nearest Neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.
- [18] Nasim Soltani, "VERITAS code and datasets." <https://github.com/nasimsoltani/VERITAS/>.
- [19] M. Belgiovine, K. Sankhe, C. Bocanegra, D. Roy, and K. Chowdhury, "Deep Learning at the Edge for Channel Estimation in Beyond-5G Massive MIMO," *IEEE Wireless Communications Magazine*, pp. 1–7, 2021.
- [20] M. Elwekeil, T. Wang, and S. Zhang, "Deep Learning for Environment Identification in Vehicular Networks," *IEEE Wireless Communications Letters*, vol. 9, no. 5, pp. 576–580, 2019.
- [21] I. Saffar, M. L. A. Morel, K. D. Singh, and C. Viho, "Deep Learning Based Speed Profiling for Mobile Users in 5G Cellular Networks," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, IEEE, 2019.
- [22] 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.
- [23] J. Yang, K. Zhou, Y. Li, and Z. Liu, "Generalized Out-of-Distribution Detection: A Survey," *arXiv preprint arXiv:2110.11334*, 2021.
- [24] J. Liu, T. Oyedare, and J.-M. Park, "Detecting Out-of-Distribution Data in Wireless Communications Applications of Deep Learning," *IEEE Transactions on Wireless Communications*, vol. 21, no. 4, pp. 2476–2487, 2021.
- [25] J. Robinson and S. Kuzdeba, "Novel Device Detection using RF Fingerprints," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0648–0654, IEEE, 2021.
- [26] DARPA, "Radio Frequency Machine Learning Systems." <https://www.darpa.mil/program/radio-frequency-machine-learning-systems>.
- [27] 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.
- [28] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A Unified Embedding for Face Recognition and Clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.
- [29] Y. Sun, Y. Ming, X. Zhu, and Y. Li, "Out-of-Distribution Detection with Deep Nearest Neighbors," in *International Conference on Machine Learning*, pp. 20827–20840, PMLR, 2022.
- [30] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, "Supervised Contrastive Learning," *Advances in neural information processing systems*, vol. 33, pp. 18661–18673, 2020.
- [31] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On Calibration of Modern Neural Networks," in *International conference on machine learning*, pp. 1321–1330, PMLR, 2017.
- [32] L. Van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of machine learning research*, vol. 9, no. 11, 2008.
- [33] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Large Batch Optimization for Deep Learning: Training BERT in 76 minutes," in *International Conference on Learning Representations (ICLR)*, 2020.