Received 11 March 2024; revised 13 April 2024; accepted 24 April 2024. Date of publication 1 May 2024; date of current version 23 May 2024.

Digital Object Identifier 10.1109/OJCOMS.2024.3396077

Resource-Efficient Spectrum-Based Traffic Classification on Constrained Devices

DAVID GÓEZ[1], ESRA AYCAN BEYAZIT[1], LUIS A. FLETSCHER[2], JUAN F. BOTERO[2], NATALIA GAVIRIA[2], STEVEN LATRÉ[1], AND MIGUEL CAMELO[1]

[1]Department of Computer Science, University of Antwerp—imec, 2000 Antwerp, Belgium

[2]Departamento de Ingeniería Electrónica, Universidad de Antioquia, Medellín 050010, Colombia

CORRESPONDING AUTHOR: D. GÓEZ (e-mail: GermanDavid.GoezSanchez@uantwerpen.be)

This work was supported in part by the Antwerpen IDLab Group; in part by the Universidad de Antioquia GITA Group; in part by the European Union's Horizon 2020 Research and Innovation Program under Grant 101017109 (DAEMON); in part by the Colombian Ministry of Science Technology and Innovation (Minciencias); and in part by the Communications Regulation Commission (CRC) under Contract CT 80740-035-2022.

ABSTRACT Traffic Classification (TC) systems are designed to identify the applications generating network traffic. Recent advancements in TC leverage Deep Learning (DL) techniques, surpassing traditional methods in complex scenarios, including those with encrypted traffic. Notably, state-of-the-art DL-based TC systems have been developed for wireless networks using Physical Layer (L1) packets. This approach overcomes the common limitation in TC research that assumes traffic flows within a wired network under a single network management domain. Despite their benefits, DL-based TC systems often demand significant computational resources, typically available only in cloud environments. Consequently, deploying models at the edge is often infeasible due to their resource-intensive nature, given their original training and optimization for high-resource environments. The inherent challenge lies in adapting these systems for edge computing scenarios, including deployment at access points. In this paper, we propose a novel methodology that exploits expert knowledge in combination with recent advances in Multi-Task Learning (MTL) and Deep Neural Network (DNN) optimization to allow spectrum-based TC systems to run on constrained devices. This paper propose a well-defined and innovative methodology for resource-efficient, spectrum-based TC to address this issue, combining MTL with DNN optimization techniques. Performance evaluations on an NVIDIA Jetson TX2 demonstrate that our most optimized MTL model, handling four TC tasks, can reduce memory requirements by a factor of 2.65x and improve execution time by 3.6x compared to sequential execution of four Single-Task Learning (STL) models in a server-grade configuration, with minimal accuracy impact (less than a 0.5% drop) and energy efficiency of 0.97 millijoules per sample at inference. Compared to other edge platforms such as the Raspberry Pi model 3B+ (RPI3B+) with a low-power Artificial Intelligence (AI)-accelerator such as the Coral Tensor Processing Unit (TPU), the NVIDIA Jetson achieves a 12-fold improvement in energy efficiency with no impact on accuracy.These are the first available results to provide a benchmark for different performance metrics (memory, computing, energy) over heterogeneous constrained devices for this type of TC system.

INDEX TERMS Artificial intelligence, deep learning, multi-task learning, power consumption, energy efficiency, parallel computing, IQ samples, traffic classification, AI accelerator.

I. INTRODUCTION

IN THE modern era, wireless communication systems have become a cornerstone of global connectivity, pivotal in connecting many devices ranging from smartphones to emerging Internet of Things (IoT) technologies [1]. As these systems evolve to accommodate a growing number of devices and data-intensive applications, the challenge of efficiently managing spectrum resources while ensuring optimal Quality

of Service (QoS) becomes increasingly significant [2]. One critical aspect in this context is the use of TC systems to understand network traffic behavior. These systems enable the correlation of traffic patterns with bandwidth and latency requirements and facilitate enforcing specific security and QoS policies [1], [3], [4].

Traditional TC systems primarily operate at the network or application layer (byte/protocol representation), employing various techniques such as Deep Packet Inspection (DPI), port-based analysis, and statistical Machine Learning (ML)-based flow analysis [4]. However, these methods have faced significant challenges in performance, scalability, privacy concerns, and the ability to handle encrypted traffic [5]. DL-based TC systems have surpassed such traditional methods and can be considered the state-of-the-art approach to designing them [5], [6]. However, TC systems often assume that traffic belongs to the same network domain and utilize a byte/protocol representation at the Link Layer (L2) or above, typically in a wired network environment. Although this system is efficient in wired environments, it also faces limitations when dealing with heterogeneous and complex settings like wireless networks.

A new generation of DL-based TC systems operating at the spectrum level has emerged in response to the abovementioned limitations [7], [8], [9]. Analyzing network traffic using L1 packets offers a unique perspective, allowing the classification of traffic types based on their spectral signatures, encryption independence, network domain, or the technology generating the L1 packets. However, it has been shown that the resulting DL models are large and complex, requiring high-end capacity hardware for deployment and execution. This is primarily due to L1 packets, in contrast to L2 or higher packet representations, which undergo modulation, coding, and sometimes encryption before transmission. Consequently, transmitting identical user L2 packets can lead to varied spectral representations.

Consider, for example, the models presented in [7], which demanded high-performance accelerators such as the Tesla V100[1] Graphics Processing Units (GPUs) (5120 Compute Unified Device Architecture (CUDA) cores and 32GB Random-Access Memory (RAM)) or GTX 1080 Ti[2] GPUs (3584 CUDA cores and 11GB RAM) for both model training and achieving real-time inference. These models proved impractical for real-time execution on laptop-grade GPUs like the GTX 1650[3] due to limited memory (4GB), allowing only a small number of samples to be batched for inference. Additionally, the low number of CUDA cores (1024) contributes to increased inference time per sample. Furthermore, even with a Data Center (DC)-grade server, scalability is hindered as the resources required to run these TC systems are proportional to the number of models running in parallel.

[1]https://www.nvidia.com/en-gb/data-center/tesla-v100/

[2]https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1080-ti/specifications/

[3]https://www.nvidia.com/en-eu/geforce/gaming-laptops/gtx-1650/

MTL refers to a type of ML where multiple learning tasks are simultaneously solved [10], providing the advantage of shared knowledge among these tasks. This approach proves particularly effective in complex domains like TC, where different yet related tasks can mutually inform and improve each other. Recently, state-of-the-art TC systems have embraced MTL to facilitate TC at the edge, allowing a single model to both classify traffic and predict future traffic loads [4], [11], [12]. In this approach, most of the layers used for feature extraction are shared, with only the final layers (one branch for the classifier and one for prediction) remaining independent. However, this may not suffice to ensure the real-time performance of TC systems using L1 packets beyond DC-grade computing hardware. Furthermore, in the field of Signals Intelligence (SIGINT), which relies on raw spectral data, the state-of-the-art has predominantly focused on optimizing STL DNN models for inference on constrained devices [13], [14], specifically in classification tasks such as Automatic Modulation Classification (AMC) [15], [16] and Technology Recognition (TR) [17].

To perform TC at the edge or beyond (e.g., at the Access Point (AP) itself), novel approaches to achieving TC at L1 that are both resource-efficient and capable of operating under hardware constraints require a tailor-made design. This design integrates expert knowledge (e.g., selecting the most suitable layer for feature extraction based on the input data), the model architecture (STL vs. MTL), and the optimization required based on the target device and Key Performance Indicator (KPI) for inference (e.g., inference time).

This study proposes a novel methodology for designing TC systems operating at the spectrum level, tailored for devices with constrained computational resources under 15 Watts. The given power range aligns with the typical power consumption in edge and IoT APs [18], [19]. The main contributions of this paper are summarized as follows:

1) A methodology that leverages the latest advancements in MTL and lightweight ML algorithms to provide an efficient and effective solution for this problem. To the best knowledge of the authors, this is the first work proposing a detailed methodology to design tailor-made DNN for TC at the spectrum level, providing both STL and MTL models that can run on constrained devices such as the NVIDIA Jetson TX2,[4] removing the limitations of state-of-the-art works that are resource hungry such as our previous one [7].
2) The design of an optimized MTL architecture on Convolutional Neural Network (CNN) and Dynamic Task Prioritization (DTP) training for TC using L1 packets for constrained devices. Compared to the state-of-the-art of byte-based TC [12], [20] and SIGINT-related tasks, e.g., AMC using In-phase and Quadrature components (IQ) samples [21], our work also addresses the problem of balancing the learning across different tasks.

[4]https://developer.nvidia.com/embedded/jetson-tx2

3) Extensive experimentation is provided to assess the flexibility of the proposed methodology in creating an optimized MTL model for TC at the spectrum level. The resulting model can run four TC tasks on a constrained device like the NVIDIA Jetson TX2. More specifically, our most optimized MTL model, handling four TC tasks, can reduce memory requirements by a factor of 2.65x and improve execution time by 3.6x compared to the sequential execution of four STL models in a server-grade configuration, with less than a 0.5% drop in accuracy, ensuring a performance aligned to what it is expected for real-time TC according to [7].
4) Performance evaluation and analysis of the energy (joules) and power (watts) consumption, together with the energy efficiency (joules/sample) of the resulting optimized MTL model, are obtained when running on three different edge hardware platforms: the NVIDIA Jetson TX2, a RPI3B+,[5] and a Coral TPU USB Accelerator.[6] The results show that the NVIDIA Jetson TX2 achieves energy efficiency for TC at 0.97 millijoules per sample, marking a 12-fold improvement over the RPI3B+ when leveraging the Coral TPU as an AI accelerator while maintaining real-time execution.The results indicate that the Jetson TX2 achieves energy efficiency for TC equivalent to 0.97 millijoules per sample, representing a 12-fold improvement in energy efficiency compared to the RPI3B+ when utilizing the Coral TPU as an AI accelerator and while still ensuring real-time execution. Combined with the previous contribution, these results represent the first comprehensive benchmark for diverse performance metrics (memory, computing, energy) across various constrained devices for this type of TC system.

The rest of the article is structured as follows. Section II provides an overview of related work. Section III presents a two-step methodology for spectrum-based and resource-efficient TC, with design and implementation details about the first and second phases presented in Sections IV and V, respectively. Section VI discusses the performance evaluation of the optimized and non-optimized STL and MTL models resulting from our methodology on constrained devices such as the NVIDIA Jetson TX2 and RPI3B+ with and without AI accelerators such as the Coral TPU. Finally, Section VII summarizes our conclusions and outlines future work. For the convenience of readers, Table 1 lists the acronyms used in this paper.

TABLE 1. List of acronyms used in this paper.

Acronym	Description
AE	Autoencoder
AI	Artificial Intelligence
AMC	Automatic Modulation Classification
AP	Access Point
CNN	Convolutional Neural Network
Conv1D	1D Convolutional layer
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAF	Discrete Autocorrelation Function
DC	Data Center
DCI	Downlink Control Information
DL	Deep Learning
DNN	Deep Neural Network
DPI	Deep Packet Inspection
DSSS	Direct-Sequence Spread Spectrum
DTP	Dynamic Task Prioritization
DWA	Dynamic Weight Averaging
ETC	Estimate-Then-Classify
FFT	Fast Fourier Transform
GL	Gossip Learning
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HDLC	High-level Data Link Control
HPS	Hard Parameter Sharing
IoT	Internet of Things
IQ	In-phase and Quadrature components
KD	Knowledge Distillation
KPI	Key Performance Indicator
KPImtl	KPIs from the MTL
KPIopt	KPIs from the optimized model
KPIstl	KPIs from the STL
L1	Physical Layer
L2	Link Layer
L7	Application Layer
LSTM	Long Short-Term Memory
MCS	Modulation and Coding Scheme
MGDA	Multiple Gradient Descent Algorithm
ML	Machine Learning
MTL	Multi-Task Learning
MTT	Multi-Task Transformer
NIN	Network In Network
NMS	Network Monitoring Services
OFDM	Orthogonal Frequency Division Multiplexing
ONNX	Open Neural Network Exchange
PHY	Physical Layer
QoS	Quality of Service
QPSK	Quadrature Phase Shift Keying
RAM	Random-Access Memory
RAT	Radio Access Technology
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
RPI3B+	Raspberry Pi model 3B+
SDR	Software Defined Radio
SHLNN	Single-Hidden-Layer Neural Network
SIGINT	Signals Intelligence
SPS	Soft Parameter Sharing
STL	Single-Task Learning
TC	Traffic Classification
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
TR	Technology Recognition
WPA	Wi-Fi Protected Access

II. RELATED WORK

This section introduces key works in TC, covering spectrum representation, MTL approaches, and optimized DNN architectures tailored for computational efficiency in TC. For an in-depth exploration of ML/DL approaches for TC, we recommend referring to [3] and [22]. Those interested in MTL can find valuable insights in [10], and for optimization methods focusing on DNN in constrained devices, [23] provides a comprehensive resource.

[5]https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/
[6]https://coral.ai/docs/accelerator/datasheet/

A. TC AT ANY LAYER

Several works, including [7], [9], [24], [25], have identified limitations in byte-based approaches when applied to wireless networks. In response, recent years have seen the emergence of spectrum-based TC systems that work on raw spectrum data. A pioneer work in this domain is presented in [24]. The authors proposed a classification algorithm based on a Discrete Autocorrelation Function (DAF) that uses binary information collected by spectrum sensing to identify the pattern traffic of the primary user. The proposed algorithm identified traffic patterns as stochastic and deterministic ones. Another approach was presented by Liu et al. in [26], which utilizes a similar input data format but employs more advanced classifiers, such as Estimate-Then-Classify (ETC), in order to remove the assumption from previous works of having a perfect period measurement of the primary user activity.

Testi et al. have proposed using ML classifiers to directly identify YouTube and WhatsApp applications from spectrum data [25]. Input data for these algorithms comprises four specific features (mean, variance, kurtosis, and packet rate) extracted from 5-second captures of IQ samples. The Single-Hidden-Layer Neural Network (SHLNN) has achieved an accuracy exceeding 97%.

A different approach was taken by theauthors in [9], [27], who introduced a CNN-based TC algorithm using images representing snapshots of the radio spectrum with an accuracy of up to 96% using a synthetic dataset. However, these approaches assume that IQ samples come from single-user and single-flow scenarios, which requires a mechanism to identify the flows directly on the spectrum. Girmay et al. [28] tackled this limitation by proposing a traffic characterization process where the output of an IQ-based TR module is used to identify the traffic characteristics of the technologies in terms of channel occupancy time, transmission pattern, and frame count using binary representations, a representation similar to the one used in [24] and [26]. The obtained results showed that the proposed solution can be used to characterize the identified traffic effectively.

An alternative to overcome the limitations discussed in [9], [27] is to leverage raw L1 packets, e.g., based on IQ samples, and conduct TC directly on them. Using a Recurrent Neural Network (RNN) architecture, the authors in [29] demonstrate that TC on raw spectrum data can be achieved using short time series (a few hundred samples) with an accuracy of $\leq 85\%$. While this accuracy might seem low compared to byte-based TC systems, it is important to note that L1 packets were single-modulated with no coding, non-encrypted, and transmitted with a low data rate. As experimentally demonstrated later in [7], one contributing factor to this performance could be the use of RNN architectures, such as Long Short-Term Memory (LSTM) [30], [31], which are known to face challenges in terms of inefficient training and achieving high accuracy with large data sequences [32], [33], [34].

More recently, [7] have proposed two DNN-based classifiers, a novel 2D-CNN spectrum-based TC and a Gated Recurrent Unit (GRU) as baseline architecture, and have benchmarked their performance on three TC tasks at different protocol layers. The performance evaluations show that the 2D-CNN model can achieve an accuracy above 92% in the most demanding TC task, with only a 4.37% drop in accuracy compared to a byte-based DL approach. The model exhibits microsecond per-packet prediction time on server-grade hardware, which is very promising for delivering real-time spectrum-based traffic analyzers.

B. MTL FOR TC

The authors in [4] highlight that most research in Network Monitoring Services (NMS) predominantly concentrates on STL. This means each model is specifically developed and trained for a distinct task, such as TC, traffic prediction, or anomaly detection. To address this limitation, MTL strategies are suggested. In [4], the authors employ an MTL framework to concurrently address TC and traffic prediction using a two-step process with Autoencoder (AE). They use traffic data that includes Downlink Control Information (DCI) messages with a detailed time granularity of 1 millisecond.Compared to conventional STL approaches, which did not use AE and tackle classification and prediction tasks separately, the MTL approach always provided the highest performance.

Another MTL model for TC has been presented in [35]. In their framework, a CNN-based model using statistical features is developed to solve tasks such as traffic classes, bandwidth requirements, and duration of traffic flows. The statistical features are packet length, inter-arrival time, and packet direction. The experiments have demonstrated that, even with a reduced amount of labeled data, the classification accuracy remained high, underscoring the effectiveness of MTL in situations with limited data availability. Similar conclusions were provided by [36], where MTL trained with only 150 labeled samples can emulate the 94.67% accuracy achieved through STL with 6139 labeled samples.

A distributed approach for MTL has been presented in [20], where Gossip Learning (GL) is used to exchange peer-to-peer information during training.The proposed LSTM-based model uses 84 transmission-related features from different protocol stack layers to feed a shared AE, which acts as a feature extractor and then connects to dense (fully-connected) layers, which act as predictors. The results show that the distributed MTL approach performs similarly but saves energy concerning their correspondent centralized versions and benchmark solutions.

Authors in [12] presented the DISTILLER classifier, which adopts a multi-modal MTL approach for encrypted TC. It simultaneously utilizes heterogeneous inputs to address multiple related classification tasks, supporting various application scenarios with diversified network visibility. The DISTILLER architecture incorporates single-modality layers for the payload and protocol field modalities, including

1D convolutional layers, bidirectional GRU, and dense layers. Intermediate features from these modalities are merged and fed into the shared representation and task-specific layers, with the outputs obtained via Rectified Linear Unit (ReLU) activations. The results showed that DISTILLER outperformed the baselineMTL architectures.

More recently, a Multi-Task Transformer (MTT) model has been proposed to jointly address application identification and traffic characterization tasks [37]. In MTT, the input packet is represented as a sequence of bytes and utilizes a multi-head attention mechanism to extract features. This approach is notable for being the first to introduce transformers into the multi-task classification of network traffic.Experimental results have shown that the MTT model efficiently produces both outcomes in approximately 0.1 milliseconds per packet, meeting the demands for real-time online TC with an F1 score above 98% on both tasks, outperforming previous state-of-the-art work.

Authors in [38] combined a CNN-based transformer with meta-learning to avoid the costly task of model retraining and enable out-of-distribution traffic sample classification. Different from the previous approach, the authors first split the raw traffic files by session, and then convert each packet into a fixed-format gray-scale image as input to the model. The performance evaluations showed that the proposed architecture outperformed other baselines based on DNNs and transformers in terms of both accuracy ($\leq 93\%$) and lower inference time (0.96ms).

C. OPTIMIZED DNN FOR TC

In recent years, optimization techniques have been proposed to reduce the complexity of DNN-based architectures for TC at L2 or above. At the architectural level, authors in [39], [40], and [37] have included attention mechanisms to reduce the number of hidden layers since these mechanisms help to process only relevant subsets of high-dimensional inputs and to focus on the most pertinent aspects of the data. In all cases, the models outperform the state-of-the-art baselines in terms of accuracy while improving execution time (up to 50% reduction) with similar model sizes.

Other approaches include model optimization using compression techniques [23]. In [41], Lu et al. propose a compressed Network In Network (NIN) model for TC. They design a step-wise pruning and Knowledge Distillation (KD) strategy to train the compressed model, aiming to reduce storage and computing resources. The resulting model achieved a 50% reduction in model size with up to a 30% improvement in computation time compared with the uncompressed NIN model. In terms of accuracy, the models achieved an average F1 score of 98.05%, surpassing that of the CNN state-of-the-art model used as a baseline. Although the experimentation did not evaluate the model on a constrained device, the calculated Tera Operations Per Second (TOPS) of their best model would be suitable for constrained devices.

The NIN basic architecture is optimized in a follow-up work using self-distillation and KD for TC [42]. The model is further optimized with pruning to remove redundant filters and employs knowledge distillation to train compressed models without compromising performance. Performance evaluation showed that the model could achieve a processing time of less than 0.023 ms/sample with nearly a 99% computational overhead reduction compared to the baseline. Although the results seem promising, no validation has been provided on constrained devices.

D. RESEARCH GAPS AND POSITION OF THIS WORK IN THE LITERATURE

In this paper, we exploit advances in MTL techniques and DNN optimization via a novel methodology to offer an efficient and effective solution for TC that covers the three previously mentioned dimensions: classification at any layer, MTL support and model optimization to run on constrained devices, as summarized in Table 2.

In the first dimension, we continue using L1 packets to perform the **TC at any layer** from our previous work [7] since this approach has demonstrated competitive performance compared to byte-based TC. Compared to recent works such as [28], which employ DL techniques to perform traffic characterization at flow level directly on the spectrum, L1 packet-based TC still provides more flexibility in classifying traffic types at any layer and granularity.

Concerning MTL, this is the first work introducing a novel and flexible methodology to generate optimized models for both STL and MTL architectures for TC using L1 packets to the best of the authors' knowledge and going beyond our previous work [7]. Moreover, our resulting MTL 1D-CNN model was trained with DTP, which also addresses the challenge of balancing learning across multiple tasks. Notice that although MTL has shown very promising performance in terms of accuracy and inference time for real-time deployment of byte-based TC systems [37], [38] and helps to mitigate the resource-intensive nature of DNNs [43], this approach is still not sufficient to support edge deployments of spectrum-based TC with L1 packets. This aspect triggers the integration of the third dimension in our proposal.

In STL and MTL approaches for TC, most of the architectures are not *optimized to run on constrained devices*, except very recent state-of-the-art TC based on transformer architectures with attention mechanisms [38] or DNN compression [42] but with no results on their performance on actual hardware. Our approach exploits recent advances in DNN optimization to achieve such a goal. In addition to being the first work on combining MTL and DNN optimization techniques to allow TC using L1 packets, we also differ from previous works since we provide extensive experimental validation on different resource-constrained platforms and benchmark the performance of the generated MTL models in terms of memory, computing, inference time, and energy efficiency on these platforms.

TABLE 2. A comparison of the analyzed related work and how this work is positioned in the literature.

Paper	Classification at any layer	Input representation	MTL	Proposed Model	Optimized for constrained devices
[24]	Yes	Binary time series representing spectrum occupancy	No	DAF	No
[26]				ETC and others	
[9], [27]		Raw IQ samples		LSTM + softmax	
[28]		FFT from RAW IQ samples		2D-CNN	
[29]		Raw IQ samples			
[7]		Images representing sequence of RAW IQ samples			
[26]		Mean, variance, kurtosis, and rate of packets		LR, SVM, SHLNN	
[4]	No	Transport Block Size	Yes	LSTM+softmax, LSTM+dense layer	
[25]		Packet length, inter-arrival time, packet direction		1D-CNN	
[20]		84 features related to all the layers of LTE protocol stack		LSTM+softmax, LSTM+dense layer and GL for distributed training	
[12]		Bytes from L4 payload, header, and sequence of packets		1D-CNN and BiGRU	
[36]		Traffic identifer, packet size, and inter-packet interval of the first 80 packets		1D-CNN, GRU, LSTM, and sparse AE	
[37]		1500 bytes from a pre-processed L2 packet		1D-CNN with attention mechanism	No, but the attention mechanism reduces its complexity compared to traditional DNN without them
[38]		Images representing sequence of byte-based packets		2D-CNN with attention mechanism	
[39]		Images representing 100 bytes of headers at different layers	No	LSTM with attention mechanism	
[40]		Combination of packet flow statistics and raw packet.		CNN with attention mechanism	
[41]		1500 bytes from a pre-processed L2 packet		Compressed NIN model via KD	Yes, but not validation on constrained hardware
citeLu2023CompNN				Compressed NIN model via two-step distillation procedure	
This study	***Yes***	***Raw IQ samples***	***Yes, optimized with DTP***	***1D-CNN***	***Yes, Nvidia Jetson TX2, RPI3B+, and Coral TPU***

The proposed methodology, discussed in the following sections, allows the generation of L1-based TC systems that are resource-efficient and capable of functioning within hardware limitations via customized design. This design integrates expert knowledge, such as selecting the optimal feature extraction layer based on input data, choosing

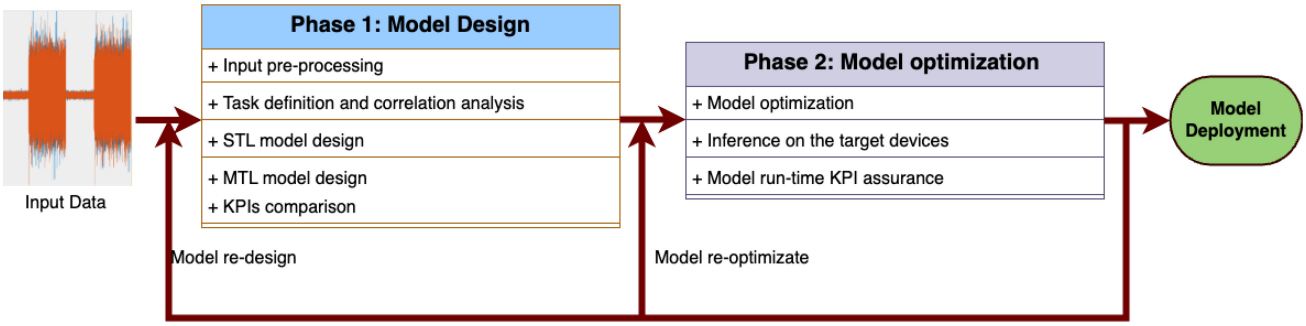

FIGURE 1. Methodology for designing spectrum-based TC systems capable of running on devices with limited resources.

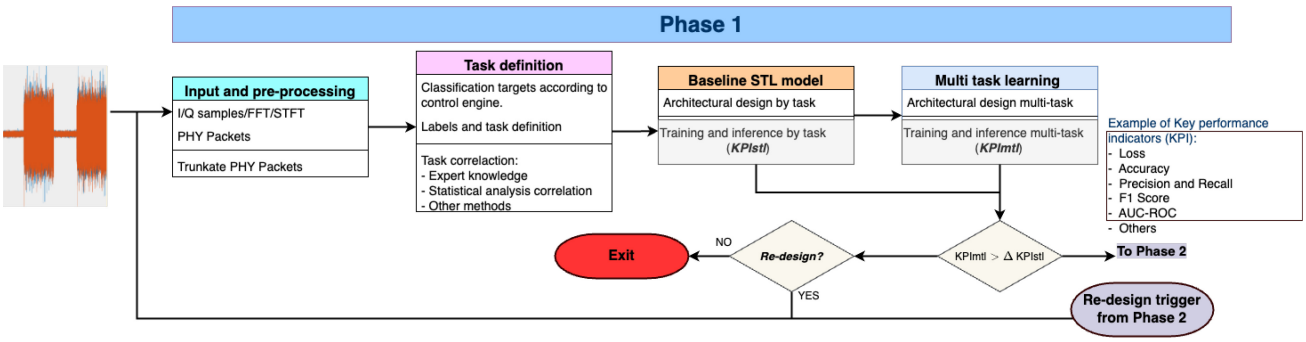

FIGURE 2. Composition of Phase 1: a four-step process from input pre-processing to models ready for optimization.

between STL and MTL architectures, and optimizing for specific hardware constraints and KPIs such as inference time and energy efficiency.

III. A RESOURCE-EFFICIENT METHODOLOGY FOR SPECTRUM-BASED TC

In this section, we will introduce a novel and well-defined methodology to achieve a resource-efficient TC using L1 packets that can run on constrained devices. Figure 1 shows the proposed two-phase methodology that combines MTL and DNN optimization mechanisms tailored to a target device. Let us start with a general description of each phase.

A. PHASE 1: PRELIMINARY DESIGN

In this phase of the proposed methodology, we intend to identify if we can solve multiple tasks with STL models via MTL or, if not possible, optimize each STL model individually during the second phase. The first phase is composed of five specific steps, as summarized in Figure 2:

- *Input pre-processing:* Similar to [7], a sequence of IQ samples is collected to create truncated L1 packets. Other representations of the truncated packets, such as Fast Fourier Transform (FFT) as raw floating points or images, can also be used.
- *Task definition and correlation analysis:* Task correlation is established among tasks using expert knowledge, statistical analysis, and other methods using the training dataset to understand how tasks are related and can mutually benefit from sharing knowledge while training.
- *STL model design:* Baseline models on each task are designed to measure the KPIs from the STL (KPIstl) models, such as accuracy or F1-score, training or inference time, and memory consumption. These values will be used later as a reference to compare with the results of the MTL model. Notice that if there are no tasks with high correlation in the previous step, these models are used as input in phase two.
- *MTL model design:* An MTL model will be created if the task correlation step finds tasks with high correlation. Performance evaluations measure the KPIs from the MTL (KPImtl) model during training and inference. These KPIs are expected to be the same (or close to) as the KPIstl per task models for further comparison.
- *KPIs comparison:* If the KPImtls are equal or close to those of the individual tasks (KPIstl), we will proceed with the second phase with the trained MTL model. Otherwise, a re-design of the model will be considered.

At the end of this phase, we will have either a set of STL models or an MTL model for further optimization. We aim to provide the designer enough flexibility to solve one or more related tasks for TC using raw spectrum data. In the case of an MTL model, it is important that the KPImtls, which can be decomposed into KPIs per task, are the same or very close to the ones of the STL models in a given Δ-tolerance. If this is not the case, a re-design or a termination step is followed.

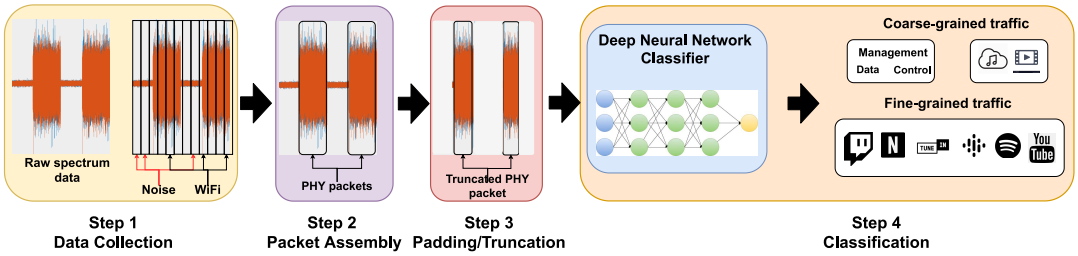

FIGURE 3. End-to-end TC system using spectrum data, which comprises four steps: 1) data collection, 2) L1 packets filtering/assembly, 3) zero padding or data truncation of the time series, 4) Fine- or coarse-grained traffic classification. Steps 1 to 3 are examples of the input and pre-processing part of the first phase.

B. PHASE 2: OPTIMIZATION AND DEPLOYMENT

In the second phase, model optimization mechanisms are applied to ensure that the final model can satisfy all the constraints of the specific devices on which it will be deployed and run. Depending on the output from the first phase, i.e., multiple STL models vs. single MTL model, how the model(s) will be deployed and executed for inference will be defined.

This phase is composed of three main steps, as summarized in Figure 2:

- *Model optimization:* Depending on the output of the previous phase (STL vs. MTL model) and the target device, different DNN model optimization mechanisms, such as DNN pruning, down-sampling, quantization, precision-reduction, layering, and feature-combining techniques, can be applied. For example, precision reduction techniques can be applied if the target device does not support integer 8 or 16 quantization. In addition, the number of hardware resources on the target device will indicate how STL models can be executed since very constrained devices will run a maximum of one model at a time, while less constrained ones may run more than one in parallel. In the case of an MTL model, the architecture can run the inference in parallel.
- *Inference on the target device:* Once the model is optimized and can run on the target device, performance evaluations are carried out to obtain the model's KPIs from the optimized model (KPIopt). After the optimization, some performance degradation may be expected compared to the original model(s). However, it should again fall in the Δ-tolerance zone compared to the KPIstl or KPImtl. The value Δ will depend on the task and the KPIs that the designer can trade-offs, e.g., model size vs. inference time. At this point, the resulting model(s) can run on the target device.
- *Model run-time KPI assurance:* This step is related to constraints during run-time. Some tasks may be part of near real-time decision-making processes (e.g., $< 1s$) or slow ones (e.g., $\geq$ 1s). For example, if the arrival time of packets is 1000 packets/sec, then $exec_t$ per packet must be less than $max_{exc_t} = 1$ ms. Once the model fulfills the run-time KPI, it can then be deployed in production. If the model(s) can not run on the target device, then a re-optimization (e.g., using other parameters, techniques, or hardware), re-designing (smaller architecture), or exit steps will follow.

As described above, the methodology provides a well-defined step-by-step model design to create spectrum-based TC systems targeting deployment on constrained devices. While the set of steps is well-defined, some of them allow enough flexibility to target different trade-offs between learning and run-time KPIs.

IV. PHASE 1: PRELIMINARY DESIGN

In this and the next sections, we validate the proposed methodology through an end-to-end design and experimentation of an MTL model capable of performing the three TC tasks evaluated in [7], along with an additional task, on an NVIDIA Jetson TX2, an AI accelerator with power consumption less than 15 Watts. For comparison, a Tesla $V100$ can consume up to 300 Watts, and the GTX 1080 Ti can consume up to 250 Watts at peak performance. We first focus on Phase 1.

A. INPUT PRE-PROCESSING

As described in [7], any L1 packet can be obtained directly from the spectrum using a technology-agnostic procedure, as shown in Figure 3.

In general, the 4-step procedure can be summarized as follows. The first step is data collection, where an algorithm captures and pre-processes spectrum samples. It includes spectrum sensing, normalization, and labeling of samples based on the Radio Access Technology (RAT), noise, or interference. Once the samples are collected, we assemble and filter the L1 packets. This step involves assembling L1 packets from IQ samples collected in the first step, using labels to filter and organize these samples. Techniques like cross-correlation or ML approaches can be used to enhance the robustness of packet detection. Other approaches based on other representations, such as the FFT, can be used to change the representation of the raw IQ samples to improve the accuracy of the assembly.

Once the L1 packet is created and noise samples are discarded, the packet is either padded or truncated. This normalization improves the training and inference speed of DL models, though it increases memory requirements. The optimal length for L1 packets varies based on DL architecture and the specific RAT. The final step is the

TABLE 3. Description of the proposed classification tasks to evaluate the spectrum-based TC approach.

Task ID	Traffic Classification Task	Traffic Classification type	Input representation	Layer on which the task has meaning	*No* Classes	Classes
0	**Technology characterization**	Coarse-grained	IQ samples	L1	3	802.11b, 802.11g, 802.11n
1	**Frame characterization**	Coarse-grained	IQ samples	L2	3	Management, Control, Data
2	**Application characterization**	Coarse-grained	IQ samples	L7	3	Audio, Video, No application type
3	**Application identification**	Fine-grained	IQ samples	L7	7	Netflix, Youtube, Twitch, Spotify, Gpodcast, TuneIn, No application

classification task itself, where the DL model(s) classifies the prepared L1 packets. This classification can be at different layers, starting from broader distinctions (like separating other Physical Layer (PHY) transmission or frame types at L2) to more specific classifications at higher layers, such as identifying the type of Application Layer (L7) traffic or the originating app for the data.

Depending on this pre-processing, we can identify which DNN architecture is more suitable for solving the TC tasks. For example, raw IQ samples can be processed directly using CNN and RNN, each with its own computational cost. However, transforming the input data into images representing L1 packets is unsuitable for 1D-CNN. As we will see in the third step of this phase, we select 1D-CNN instead of 2D-CNN as in [7] to reduce model size without impacting accuracy.

B. TASK DEFINITION AND CORRELATION ANALYSIS

The dataset generated in [7] contains 802.11 standard-compliant L1 waveforms for testing spectrum-level traffic classification. The waveforms are caused by different 802.11 technologies (b, g, n), which result in further transmission schemes such as Direct-Sequence Spread Spectrum (DSSS) in 802.11b and Orthogonal Frequency Division Multiplexing (OFDM) in 802.11g/n, different types of L2 frames (management, control, and data), and multiple Modulation and Coding Scheme (MCS) according to the standard.

As described in [7, Sec. V.A], the payload carried by these L1 packets (information at L2 and above) were generated using real traces of L7 applications running on a mobile device and connected to a secured 802.11 AP with Wi-Fi Protected Access (WPA)-2 on channel 1 (2.4GHz) with 20 MHz of available bandwidth. Additionally, the dataset was captured while several other wireless devices were connected to the same AP or another APs in the same band. However, they were not under management and could be generating network traffic. This configuration offers a straightforward deployment method for acquiring authentic traffic influenced by transmissions from other wireless devices sharing the same channel.

The generated dataset also encompasses a wide array of variations in the 802.11n protocol stack, including L1 encrypted packets, MCS adaptation, L1 diversity (b, g, and/or n to accommodate legacy compatibility of the AP), and L2 packet diversity. As a result, the provided dataset is more realistic and complex than the one used in [29], which is limited to High-level Data Link Control (HDLC), a simpler L2 protocol whose unencrypted waveforms are modulated only with Quadrature Phase Shift Keying (QPSK) at a unique data rate of 1Mbps, with no other devices generating traffic.

The resulting dataset contains a single L1 packet per sample, equivalent to the expected output of step two from Figure 3, where each packet is a sequence of IQ samples. From the original dataset, four different tasks are defined: tasks 1 to 3, as in [7], in addition to a classification task related to technology characterization (task 0). Table 3 summarizes the proposed traffic classification tasks based on L1 packets. Each task can be defined as follows:

Task 0 - L1 technology characterization: In this coarse-grained task, the TC algorithm uses L1 packets to determine if the packet was transmitted using 802.11b, 802.11g, or 802.11n format.

Task 1 - L2 frame characterization: In this coarse-grained task, the TC algorithm uses L1 packets to determine if the transmitted packet is a Management, Control, or Data L2 frame in 802.11.

Task 2 - L7 Application characterization: In this coarse-grained task, the TC algorithm uses L1 packets to determine the type of application inside the transmitted packet (e.g., audio or video). As only L2 Data frames carry L7 application data, then the algorithm should also discriminate packets that do not carry data.

Task 3 - L7 Application identification: In this fine-grained task, the TC algorithm discriminates between the actual applications generating the L7 traffic.

Once the tasks are well-defined, examining their correlation to exploit MTL is essential. Different approaches, such as expert knowledge and statistical analysis, can be used depending on the tasks. In our case, we perform a statistical analysis based on class distributions as it is the most straightforward approach based on the task definition. Table 4 shows that the task 0 and task 1 labels are highly correlated as there is a near one-to-one match between labels 802.11b - Mgmt, 802.11g - Ctrl, and 802.11n - Data. Furthermore, the last label correlation (802.11n - Data) is useful for identifying the correlation between the 802.11n label for task 0, the Data label for task 1, and the labels associated with applications in tasks 2 and 3, as shown in Table 5. This analysis allows us to draw an initial conclusion that the four tasks can benefit from an MTL approach since classifying task 0 (or task 1, respectively) with high accuracy will result in high accuracy

TABLE 4. Sample distribution for L1 technology characterization (task 0) and L2 frame characterization (task 1).

Total Samples	Samples per Task Label	Technology		
		802.11b	802.11g	802.11n
466348	Mgmt: 75156	74662	494	0
	Ctrl: 250967	5340	245627	0
	Data: 72264	5030	337	134858

TABLE 5. Sample distribution for L2 frame characterization (task 1), L7 application characterization (task 2), and L7 application identification (task 3).

Total Samples	Samples per Class Task 3	Samples per Class Task 2	Frames		
			Mgmt	Ctrl	Data
140665	Spotify: 13822	Audio: 39053	0	0	39053
	Tunein: 10229				
	Gpodcast: 15002				
	Youtube: 16671	Video: 56253	0	0	56253
	Netflix: 18268				
	Twitch: 21314				
	No-App: 45359	No-App-Type: 45359	14805	30554	0

on task 1 (or task 0, respectively) and simultaneously may improve the classification performance on tasks 2 and 3 since learning tasks 0 and 1 with high accuracy will reduce the misclassifications of L2 Data packets as it is acting as a No-App label filter.

C. STL MODEL DESIGN

When targeting the tailor-made design of DNN architectures for TC at the spectrum level, it is important to clearly understand the input format so the hidden layers of the models are selected to be suitable for the input representation and use the lower number of parameters. For example, in [7] and [29], 2D-CNN and RNN have been used to process raw IQ samples. While the results demonstrate that 2D-CNN outperforms RNN in accuracy and inference time, the resulting CNN models were very large (around 3M parameters per STL model).

While it is natural to consider 2D CNN the most efficient means of processing IQ samples, as they can be treated as one-dimensional data over two channels, caution is necessary during their implementation. Let us describe how 1D and 2D convolutions are implemented in DNN.

1D Convolution: Let us s consider an input sequence $x[x_1, x_2, \ldots, x_n]$, where x_i represents the i-th element of the input and n is the length of the sequence. Suppose $f = [f_1, f_2, \ldots, f_m]$ is a filter (or kernel) of length m, with $m \leq n$. The convolution operation involves sliding the filter f over the input sequence x and computing the dot product at each position. The output of this operation, known as the feature map or convolved feature, is denoted as $c = [c_1, c_2, \ldots, c_{n-m+1}]$, where each element c_j is calculated using the formula:

$$c_j = \sum_{k=1}^{m} f_k \cdot x_{j+k-1} \tag{1}$$

In this equation, c_j represents the dot product of the filter f with a segment of the input sequence starting from position j and covering m elements.

2D-CNN: Consider an input matrix X of size $N \times M$, where N and M are the dimensions of the input image or matrix. Let F be a filter (or kernel) of size $a \times b$, with $a \leq N$ and $b \leq M$. The convolution operation involves sliding the filter F over the input matrix X and computing the dot product at each position. The output, known as the feature map or convolved feature, is a matrix C. If the stride is 1 and without padding, the size of C will be $(N - a + 1) \times (M - b + 1)$. Each element C_{ij} of the feature map is computed as follows:

$$C_{ij} = \sum_{u=1}^{a} \sum_{v=1}^{b} F_{uv} \cdot X_{i+u-1, j+v-1} \tag{2}$$

In this formula, C_{ij} is the result of the dot product of the filter F with a corresponding sub-matrix of X, starting from position (i, j) and covering an area of size $a \times b$.

The structure of each kind of convolution is aligned to their general purpose. While 2D-CNNs have been designed to work on tasks such as image classification and object recognition, 1D-CNN have been explored in tasks like time-series analysis, audio processing, and natural language processing, as it allows the network to extract and learn features from sequential data.

But, why is this important? If we analyze the implementation[7] of the 2D-CNN in [7], we can see that their filter size is not aligned with the input data (height of size one and width equal to the number of IQ samples). Instead, they are squared matrices[8] and use zero-padding to enlarge the input to fit the filter size. The resulting large filters may explain why these models require large filter sizes to perform well since the padding might dilute the meaning of the input data.

This implementation detail, which may be seen as a decision mistake, could have been avoided if the 1D-CNN model had been selected initially since this architecture expects 1D filters, which already constrain the filter size to be aligned to the expected input data. As we will see more in detail in Section VI, the impact of such a design decision is around a $10x$ reduction in the number of trainable parameters (see Table 8).

Once the STL models are created and trained, we perform inference over a test dataset to measure the KPIs used to perform the trade-off (e.g., model size vs. inference time). In our case, all four tasks are for classification, so the main expected KPIs are accuracy and F1-score to measure.

[7] https://github.com/miguelhdo/tc_spectrum/blob/main/code/python/helpers/tr_models.py

[8] https://keras.io/api/layers/convolution_layers/convolution2d/

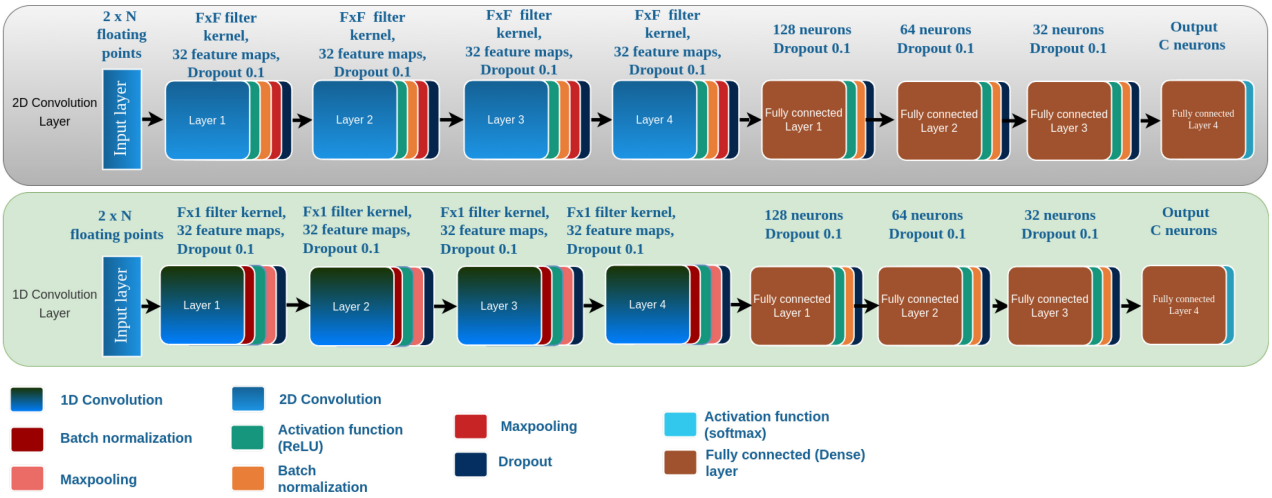

FIGURE 4. The 2D-Convolutional Layer proposed in [7] used up to 10x more parameters compares to a similar architecture with 1D convolutional layers.

These indicators will measure the learning performance of the model. Similarly, the trainable parameters of the models, such as the inference time and buffered memory (memory to load a batch of samples at runtime), can be used as indicators of the model's resource usage and runtime efficiency. In general, the learning KPIs will be a trade-off against resource usage/runtime efficiency KPIs. This set of KPIs is called KPIstl.

D. MTL MODEL DESIGN

Suppose we find several tasks highly correlated during the task definition and correlation analysis step. In that case, we will design an MTL architecture to reduce the resources required to run them in parallel. Otherwise, we will proceed to Phase 2 to optimize the STL models. In such cases, running them sequentially is always possible, albeit at the cost of an execution time that grows linearly with the number of tasks.

1) MTL STRATEGY

In the literature, several strategies for MTL exist [10], which can be grouped into two general ones [44]: Hard Parameter Sharing (HPS) and Soft Parameter Sharing (SPS). In HPS, the parameters of the convolutional layers are shared among multiple tasks. This approach allows for the extraction of features that are common to all associated tasks. The next step is to define the dense layers for each task, learning the unique details of each task. This parameter-swapping strategy significantly reduces the risk of over-fitting. A relevant aspect is its efficiency in terms of memory and calculation since fewer parameters are used compared to completely independent models for each task.

$$(W_{c1}, b_{c1}), (W_{c2}, b_{c2}), \ldots, (W_{cM}, b_{cM}) \tag{3}$$

In equation (3), we outline the architecture of the convolutional layers within a neural network. Each term (W_{cM}, b_{cM}) represents the set of weights and biases of the cM number of convolutional layers, respectively. This structure is designed to capture the input data's hierarchical patterns, leveraging the ability of convolutional layers to extract deep spatial or temporal features through convolution.

On the other hand, dense layers can be mathematically expressed as in equation (4) for the MTL model.

$$\left(W_{d1}^{t1}, b_{d1}^{t1}\right), \left(W_{d2}^{t2}, b_{d2}^{t2}\right), \ldots, \left(W_{dN}^{tK}, b_{dN}^{tK}\right) \tag{4}$$

where, $(W_{d_n}^{t_k}, b_{d_n}^{t_k})$ corresponds to the weights and biases of the n^{th} dense layer dedicated to the k^{th} task, where $n = 1, \ldots, N$ and $k = 1, \ldots, K$.

This approach enables the model to support the execution of multiple tasks by initially employing shared layers to extract features, followed by task-specific layers whose weights are finely tuned for individual tasks. On the other hand, in SPS, each task has its model with its parameters, but these parameters are regularized, so they are similar between different tasks. This is done by adding a penalty term to the loss function Equation (5). As an example, if we consider N tasks with parameters $\beta_1, \beta_2, \ldots, \beta_N$, one can represent the loss function with the SPS method as follows.

$$L = \sum_{i=1}^{N} L_i(\beta_i) + \phi \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} R(\beta_i, \beta_j) \tag{5}$$

where $L_1, L_2, \ldots, L_N$ are the loss functions for each task, R is a regularization function that penalizes the difference between the parameters of the different tasks, and ϕ is a hyper-parameter that controls the magnitude of this regularization.

It can be noticed that the number of parameters used in each approach is very different. While HPS shares physical parameters among tasks, SPS loosely shares them via the loss function. As our main target is to deploy and run large spectrum-based TC models in parallel over constrained devices, we adopt the HPS strategy. The high correlation between the selected tasks also supports this decision.

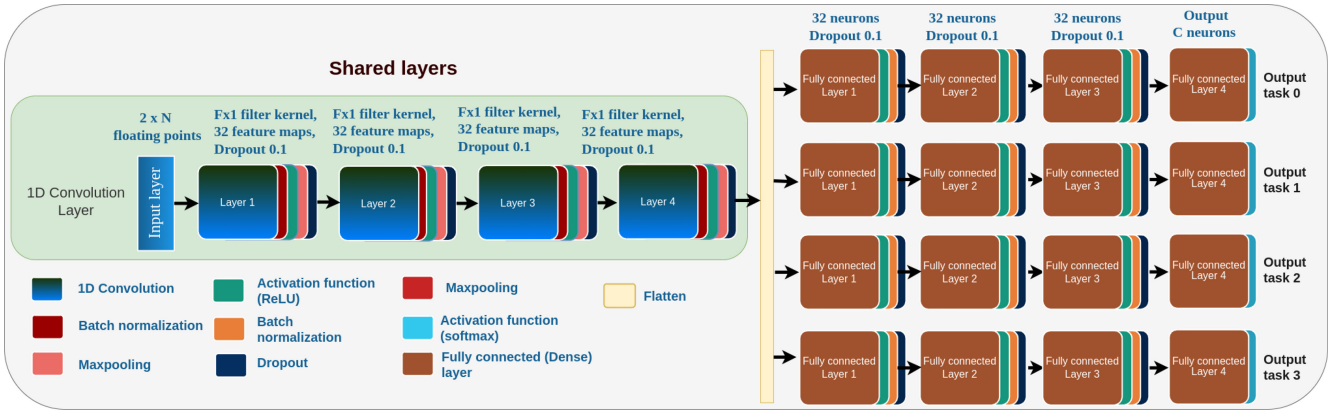

FIGURE 5. MTL architecture based on 1D-Convolutional Layer with Hard Parameter Sharing (HPS) and trained using Dynamic Task Prioritization (DTP) strategy.

2) MTL ARCHITECTURE

Once the MTL strategy is chosen, we can begin designing the architecture. As depicted in Figure 5, the proposed architecture will be based on a shared 1D Convolutional layer (Conv1D) for feature extraction among multiple tasks, followed by task-specific dense (also known as fully-connected) layers. This design enables the model to learn more specialized and unique features for each task. The resulting MTL architecture comprises an input buffer with a format of $(N, 2, 3000)$, where N is the batch size, 2 represents the in-phase and quadrature parts of IQ, and 3000 is the L1 packet truncated/padded to achieve a fixed-length input. The length of 3000 was selected based on the results obtained in [7].

The shared layers consist of four Conv1D layers, each followed by ReLU activation, batch normalization, max-pooling, and dropout layers. Next, each selected TC task has a sequence of dense layers, followed by ReLU activations and dropout layers. Each sequence has a similar structure with three linear layers, followed by ReLU activation, batch normalization, and dropout layers. One optimization we implemented here was to reduce the number of neurons on the first two dense layers, a change that minimally impacts the model's accuracy while further reducing the model size.

For classification, a final dense layer has C neurons, where C is the number of labels to classify, with a soft-max activation function to produce the output for each task. It is important to note that this MTL architecture will contain a number of trainable parameters comparable to the largest 1D-CNN STL model while being able to perform the inference of multiple tasks simultaneously. This is because the number of parameters in the dense layers across all tasks is smaller than those in the shared layers, as shown later in Table 10.

3) MTL TRAINING

One of the challenges with MTL is how much tasks differ while learning. Although tasks can be correlated, some tasks may be more challenging. This challenge can be addressed by balancing the learning across different tasks in MTL [44]. Balance between tasks is an essential stage in MTL implementations to ensure that no task dominates the learning process at the expense of the others.

Techniques like Dynamic Weight Averaging (DWA) [45], DTP [46], and Multiple Gradient Descent Algorithm (MGDA) [47] have been recently proposed in the literature. As presented in [44], DWA and DTP are techniques that focus on adjusting the weights of each task's loss based on their current training dynamics, while MGDA, conversely, looks for a joint gradient direction beneficial for all tasks. This results in MGDA being computationally more complex as it involves solving an optimization problem over the gradient space. In contrast, DWA and DTP mainly involve recalculating weights based on loss changes or predefined criteria. In addition, DTP is more flexible than DWA since it can accommodate various criteria for weight adjustment. Although DWA and DTP still require careful manual tuning of the initial hyperparameters, this is not a problem when the number of tasks is small. Therefore, we decided to implement DTP, given the extra flexibility and the limited number of tasks to learn simultaneously.

Focusing on the DTP technique, it involves adjusting the weight $\beta_i(t)$ of each task dynamically during training at each time step t, often based on each task's current performance or learning pace. In other words, if the total loss L is a weighted sum of individual task losses, then it can be expressed as:

$$L = \sum_{i=1}^{N} \beta_i(t) \cdot L_i \tag{6}$$

However, unlike DWA which is based on the change of each task's loss, $\beta_i(t)$ in DTP can be defined on different criteria, such as task difficulty, rate of improvement, or importance of the task. The calculation of the specific weight of task i in iteration t is expressed as:

$$\beta_i(t) = -(1 - k_i(t))\gamma_i \log k_i(t) \tag{7}$$

where $\beta_i(t)$ is the weight of task i and parameter $k_i(t)$ is for measuring the i^{th} task difficulty on a scale from 0 to 1. γ_i is a parameter that adjusts the task's weight according to

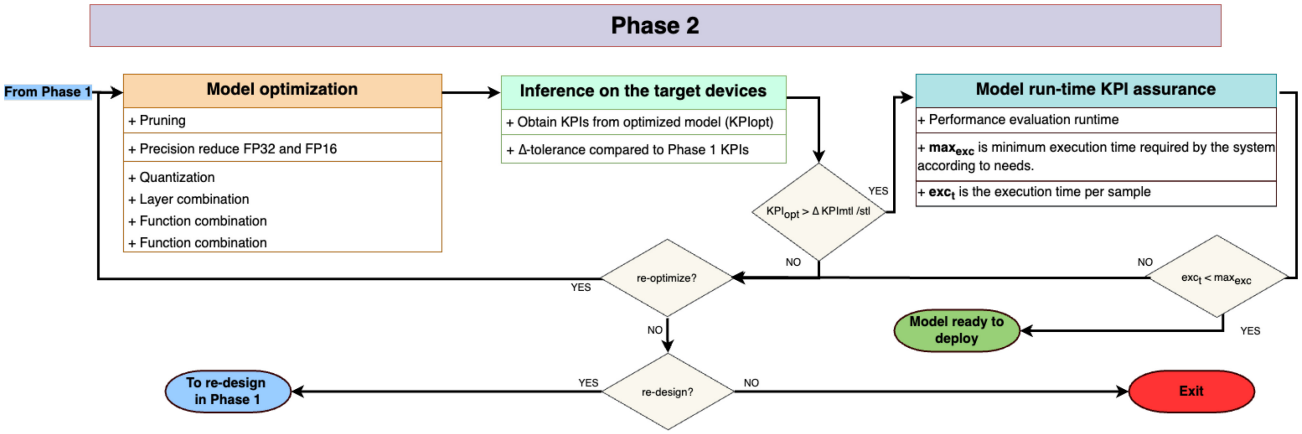

FIGURE 6. Phase 2: From model optimization to deployment.

its difficulty and $\log k_i(t)$) amplifies the differences between tasks.

To select the hyper-parameters in DTP, we focus on the learning complexity observed in tasks 2 and 3 during the STL design step, which difficulty was reflected in the achieved accuracy. This is translated into the following hyperparameter selection: i) task difficulty k_i was determined based on task progress relative to TC accuracy, and ii) γ_i was set to 1 for each task, which avoids introducing bias into the individualized learning of the tasks. Although tasks 2 and 3 may benefit from it, we did not notice any further improvement compared to k_i. As a result, the final value of β depends exclusively on improving accuracy during the learning stage.

To finalize this step, we follow a process similar to the STL design. Once the MTL model is created and trained, we perform inference over a test dataset to measure the same KPIs as in KPIstl. This set of measurements is called KPImtl. Notice that although the loss functions of each task in our MTL model are optimized jointly using DTP, we can still measure them individually. Resource usage/runtime efficiency KPIs are measured over the entire MTL. The decision to proceed to Phase 2 will depend on how far KPImtl diverges from KPIstl. The tolerance value $1 \geq \Delta \geq 0$ is used to balance them.

V. OPTIMIZATION AND DEPLOYMENT

Suppose we want to perform spectrum-based TC over modern wireless communications. In that case, we need to ensure that we can run these models as close as possible to where the data is generated and optimized to run in real-time over the limited computational resources these platforms provide [48], [49]. In this context, several works have shown the capabilities of several resource-constrained AI acceleration platforms to run optimized DNNs that work directly on spectrum data, such as AMC [15], [16], and TR [17].

For spectrum-based TC, no previous work provides such a benchmark on constrained devices. For this purpose, we select the NVIDIA Jetson TX2 module as the target device to deploy the optimized TC models. We also motivate our choice as this module has also been incorporated as an AI accelerator of state-of-the-art Software Defined Radios (SDRs) such as the AIR-T.[9] Based on this choice, Phase 2 is realized as follows.

A. MODEL OPTIMIZATION

The NVIDIA Jetson TX2 is an embedded computing platform for AI applications. While its common use has been in edge computing, recent approaches have seen this device paired with SDR-based receivers [50]. The hardware specifications are given in Table 6 and compared against a high-end computing platform used in testbeds such as GPULab.[10] We implemented the models during Phase 1 on one of the three slave servers equipped with NVIDIA GeForce GTX 1080 Ti GPU. In addition, the virtualized instance of our server has 4 Central Processing Unit (CPU) cores with 16GB of RAM. Table 6 describes both hardware platforms in detail.

It is important to note that two main aspects will drive the optimization phase depending on the selected platform. The first aspect is the hardware capabilities. For example, a Jetson TX2 module combines the quad-core ARM Cortex-A57 processor, a dual-core NVIDIA Denver2 processor, and a 256-core NVIDIA Pascal GPU in one single platform that consumes only up to 15 Watts. Compared to a GTX 1080 Ti only, this reduces up to 94% energy consumption at peak performance. Of course, there is a trade-off in the number of CUDA cores, which limits its TOPS performance. While a Jetson TX2 can achieve up to 1.33 TOPS, the GTX 1080 Ti can go up to 11.3 TOPS in FP32 precision, translating into 90% higher performance than the GTX 1080 Ti.

The second aspect concerns the frameworks for optimizing and deploying the models. For example, NVIDIA platforms provide TensorRT[11] tools, a high-performance DL inference

[9] https://deepwavedigital.com/hardware-products/sdr/
[10] https://doc.ilabt.imec.be/ilabt/gpulab/
[11] https://developer.nvidia.com/tensorrt-getting-started

TABLE 6. Hardware specifications.

Component	Specifications	
	Jetson TX2	**GPULabVirtualized Server**
CPU	ARM Cortex-A57 CPU (4 cores) + Denver2 CPU(2 cores)	4 virtualized Intel Xeon E5645 Cores
Memory type	8GB ***shared*** (CPU/GPU)128-bit LPDDR4	16/229 GB DDR3 800/1066/1333
GPU	256 NVIDIA CUDA cores (Pascal)	GeForce GTX 1080 Ti 3584 NVIDIA CUDA cores (Pascal) 11GB dedicated GDDR5X
GPU Performance	1.33 TOPS (FP32) 59.7GB/s memory bandwidth	11.34 TOPS (FP32) 484.4 GB/s memory bandwidth
Max Power	15Watts	250 Watts

tool designed to complement training frameworks such as TensorFlow, PyTorch, and MXNet. TensorRT focuses on efficiently running pre-trained networks on NVIDIA hardware. It includes an inference optimizer and runtime, offering low latency and high throughput for applications [51], [52].

As indicated before, the optimization framework expects a trained model. This model is traditionally provided in an Open Neural Network Exchange (ONNX) format and then is passed to an optimization step where different techniques are used to optimize the model aligned with the target device. Once the model is optimized, the resulting model is known as the inference engine, i.e., the in-memory representation of this trained and optimized model ready for execution. The optimized model, serialized in a file-like format, is also known as a plan within the context of model optimization using TensorRT. Finally, the execution workflow is constructed within TensorRT [53]. Figure 7 illustrates the workflow generated by these steps in the TensorRT framework for our MTL design.

One fundamental aspect of embedded platforms such as the NVIDIA Jetson TX2 compared to simple server-grade architectures is the memory module shared between the CPU and the GPU. This shared architecture allows the MTL model to run and load by directly accessing data from the shared memory, reducing data transfer costs typically incurred when memory is not shared, typical in simple server-grade architectures. In other words, this hardware architecture compensates for the limited TOPS due to the lower number of CUDA cores by reducing data transfer latency.

Now, let us focus on the optimization techniques in this step. DNN optimization techniques generally involve multiple optimization steps targeting batch processing, layer structure and grouping, and operations in the DNN model for better performance on specific hardware. Among such techniques, some of the most commonly used on different frameworks are listed below.[12]

1) *Layer Fusion:* Combines multiple layers into a single operation for improved computational efficiency.
2) *Precision Calibration (Quantization):* Adjusts computation precision (e.g., from FP32 to FP16 or INT8) to balance performance and accuracy.
3) *Kernel Auto-Tuning:* Selects the most efficient algorithms for operations based on the hardware configuration.
4) *Dynamic Tensor Memory:* Optimizes memory allocation for tensors, crucial for devices with limited memory.
5) *Weight and Activation Compression:* Compresses weights and activations to reduce model size and memory requirements.
6) *Graph Optimizations:* Analyzes and optimizes the execution graph to remove redundant operations.
7) *Multi-Stream Execution:* Enables concurrent execution of multiple inference streams to optimize GPU resource utilization.
8) *Integrated IO Memory:* Reduces memory copies between CPU and GPU for faster data transfer.
9) *Asynchronous Data Transfer and Execution:* Overlaps computation with data transfers to improve throughput.
10) *Pruning:* Removes redundant or non-essential neurons and connections from the network to reduce complexity and improve efficiency.
11) *Batch Fusion:* Merges operations across multiple input batches, enhancing execution efficiency and reducing latency for batched inference tasks.

Certain optimizations may or may not be applicable depending on the target hardware. For instance, the Jetson TX2 does not support precision calibration at INT8 precision, but it can leverage integrated IO memory due to the hardware's shared memory architecture. The model optimization process can be configured manually; however, we allow TensorRT to perform it automatically, considering the extensive range of optimization parameters and the framework's options.

Depending on the input from Phase 1 (multiple STL models vs. single MTL model), the outcome of this step is

[12]https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-803/best-practices/index.html

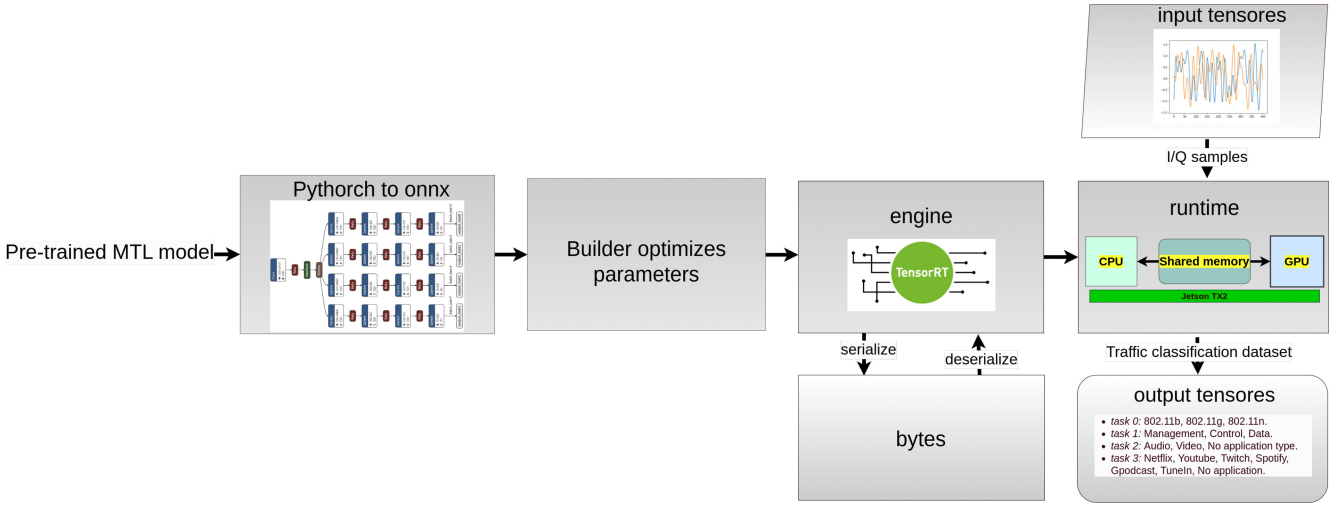

FIGURE 7. MTL Development Cycle Using TensorRT for inference.

the model optimized and capable of running on the target device. If carried optimization is insufficient to run on the target device, the designer would need to manually change the optimization parameters to achieve better compression and performance or go back to Phase 1 to re-design the original model, e.g., make it smaller or exit the process.

B. INFERENCE ON THE TARGET DEVICE AND RUNTIME KPI ASSURANCE

The final two steps are crucial for benchmarking the optimized model against the reference KPIs established in Phase 1. Specifically, verifying that the model, when executed on the target device, maintains learning and resource usage/runtime efficiency capabilities comparable to the original model after optimization is essential. The choice of benchmarks depends on whether we evaluate STL or MTL models. Accordingly, we will use either KPIstl or KPImtl. The comparison at this stage will reveal the trade-off between the original model(s) and the optimized one(s). The tolerance value, denoted as Δ, will indicate the trade-off the designer aims to achieve.

One key difference from the performance comparison in Phase 1 is that some KPIs may significantly diverge from the original model, particularly regarding runtime efficiency. Various studies have demonstrated that optimization techniques for model compression do not substantially impact model performance in learning; the optimized model can sometimes outperform the original. However, while improved learning and resource usage are anticipated, runtime efficiency largely depends on the hardware capabilities. As seen in Table 6, there is a notable difference in hardware specifications between the server-grade hardware and the Jetson TX2. These disparities can result in inference times that may be excessively long for the intended task.

Let us consider the inference time of the STL 2D-CNN model, which addresses task 2 as proposed in [7]. According to their findings, in Table 7, the 2D-CNN can classify an L1 packet containing 3000 IQ samples in just 0.15ms. In this context, this duration is significantly shorter than the maximum execution time max_{exec} of 7ms for video applications, which is necessary to ensure TC on L1 packets in real time. Notice also that although their RNN-GRU model is still shorter than the maximum execution time, their learning performance was very poor compared to the STL 2D-CNN, and it does not provide enough flexibility to include the pre-processing time of the packets (e.g., padding/truncation).

TABLE 7. Average inference time per single L1 packet in task 2 from [7].

Input length (IQ samples)	Model	Average inference time x10^{-3}
3000	CNN	0.15
	GRU-NN	5.13

As a result, the final step involves verifying that the deployed model operates quickly enough to support the real-time execution of the TC tasks. For optimized STL models, the sequential inference execution time must be shorter than the maximum allowable execution time. If the optimized models fail to achieve inference times below this maximum threshold for the task, they may need to be re-optimized or re-designed. Otherwise, the methodology concludes successfully. On the other hand, if they meet this criterion, the methodology is completed successfully, and the models are ready for deployment in production.

VI. RESULTS AND DISCUSSION

In this section, we present the evaluation results of the two phases. To complement the details about the model's implementations already discussed in previous sections, both STL and MTL models were implemented using PyTorch 2.1.[13] We use the Adam optimizer [54] with a learning rate

[13]https://pytorch.org/

TABLE 8. Comparing accuracy and average inference time per sample across 2D CNN and 1D CNN STL models from Figure 4 (Phase 1).

CNN Model		Trainable Parameters x10^3	Inference time x10^{-3}	Inference Time Aggregated x10^{-3}	Accuracy
2D	T1	3.2x10^3	0.208	0.645	0.994
	T2	3.2x10^3	0.29		0.968
	T3	3.2x10^3	0.228		0.909
1D	T1	786	0.045	**0.179**	0.995
	T2	786	0.046		0.955
	T3	786	0.046		0.896

of 0.001, a batch of size 64 (except in the last evaluations), and a cross-entropy loss function during 400 training epochs with early stopping and model checkpoint (model with best accuracy) callbacks.

The dataset used for training, validation, and testing was balanced by equalizing the number of samples across classes, with the class containing the fewest labels in task 3 (TuneIn) serving as the benchmark (see Table 5, where TuneIn has 10229 samples). Subsequently, the resulting dataset, totaling 71.6K samples, was divided into three subsets: 65% for training (47027 samples), 20% for validation (16384 samples), and 10% for testing (8192 samples). CUDA[14] v12.2 was installed on the server, while CUDA v10.2 and TensorRT v8.0.1 ran on the Jetson TX2. Our baseline model will be the 2D-CNN proposed in [7] and shown in Figure 4.

A. PHASE 1 STL AND MTL MODEL'S PERFORMANCE

One of the goals in Phase 1 is to develop models capable of addressing spectrum-based TC tasks with the fewest parameters before any advanced optimization. Let us set $\Delta = 0.95$ as the tolerance for the whole methodology. In other words, we expect the optimized models to achieve the baseline KPIstl with a maximum drop of 5%. Although the selected tolerance is very low, it holds during the methodology, as shown below.

As we explored in Section VI-C, a 1D-CNN is a more apt choice than a 2D-CNN for TC tasks using L1 packets. Despite our STL 1D-CNN having a similar architecture to the 2D-CNN, as depicted in Figure 4, it has significantly fewer trainable parameters, as indicated in Table 8. In the context of the three tasks evaluated in [7], the 1D-CNN achieves a *4x reduction in the number of the model's parameters* (3.2M vs. 786K) compared to the 2D-CNN baseline. Furthermore, this parameter reduction translates into an *average 3.6x improvement in inference time* across all tasks (0.645ms vs. 0.179ms). Finally, we can see that the accuracy among the three tasks remains almost equal, with *a minor drop of 1.3% (0.968 vs. 0.955) and 1.4% (0.909 vs. 0.896) in accuracy* for tasks 2 and 3, respectively. For the rest of the methodology, we will use the measured KPIs of the 1D-CNN as the KPIstl.

[14]https://developer.nvidia.com/cuda-toolkit

TABLE 9. Comparing accuracy of STL vs. MTL 1D-CNN models (Phase 1).

1D CNN Model	Accuracy			
	T0	T1	T2	T3
MTL w/o DTP	0.999	1	0.979	0.8852
MTL with DTP	**0.999**	0.999	**0.9734**	**0.9037**
STL	0.995	1.00	0.955	0.896

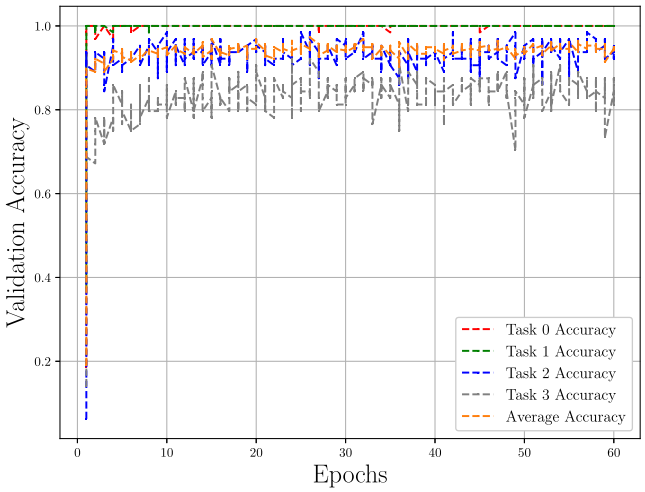

FIGURE 8. Comparison of per-task vs. aggregated accuracy of the 1D CNN MTL during training (Phase 1).

In the next step, we will develop the MTL model, utilizing the correlation analysis presented in Section VI-B. We will also include task 0 (the L1 packet TC task, as outlined in Table 3), to increase the complexity of the problem and showcase the full potential of our methodology. Figure 5 shows the trained MTL model, which uses the HPS and DTP strategies introduced in Section IV-D3. Table 9 illustrates how the MTL model performs with and without the DTP task-balancing learning strategy.

Notably, applying the DTP strategy resulted in a 1.85% increase in accuracy for task 3. This performance demonstrates that DTP can automatically identify the weights aligned with the task difficulty during the learning phase. It can be observed that the 1D-CNN MTL model has an **accuracy difference of less than 0.5% for any task** compared to the 2D-CNN (task 1 - 0.994 vs. 1.00, task 2 - 0.973 vs. 0.968, task 3 - 0.9037 vs. 0.909). These results will be part of the KPImtl.

To provide a complete picture of the MTL model's accuracy performance, Figure 8 presents the training accuracy of the 1D-CNN MTL model. Tasks 0 and 1, which are the easiest and have the highest accuracy, peaked in the initial epochs, while tasks 2 and 3 required more time to converge. Nevertheless, the model demonstrates a robust overall performance, as indicated by its high average

TABLE 10. STL vs. MTL memory requirements with a similar number of layers and neurons using a batch size of 64 samples (Phase 1).

Model		Parameters x10^3	Model size MiB	Memory Conv Layers MiB	Memory Dense Layers MiB	Total GPU Memory (Buffered memory) MiB
STL task 0	per task	203.6	0.81	185.90	0.89	186.72
STL task 1						
STL task 2						
STL task 3						
Aggregated (T0 to T3)		814.6	3.24	743.6	3.56	746.88
MTL		**784.8**	**3.14**	**185.90**	**3.3**	**189.20**

TABLE 11. Comparison of inference time for STL vs. MTL models (Phase 1).

Model	Inference Time x10^{-3}	Inference Time Aggregated x10^{-3}
Task 0	0.043	0.180
Task 1	0.045	
Task 2	0.046	
Task 3	0.046	
MTL	**0.052**	**0.052**

accuracy, represented by the orange line. Most models typically reach their optimal performance around epoch 70.

In terms of resource usage and runtime efficiency, the performance of the MTL model is given in Tables 10 and 11, which compare the model sizes (parameter count and memory size) and inference time with those of STL models. Table 10 shows that the MTL model's size (both in MiB and number of parameters) is quite similar to the aggregated size of the four STL models, being only about 3.5% larger. However, **model size plays only a minor role in the model's total memory and computational complexity**. While the MTL model requires 4*x* more storage for the model parameters (3.14 MiB vs. 0.81 MiB), this represents less than 2% of the total (buffered) memory requirements at inference time (3.14 MiB vs. 189.2 MiB).

In total, *the memory requirements for the MTL model are only 1.3% larger than those for a single STL model, which is a 4x reduction compared to the parallel (aggregated) execution of all the STL models*. Another significant observation is that the memory requirements for Conv1D layers are the predominant factor for storage, accounting for more than 98% of the total memory requirements. Consequently, a 1D-CNN MTL model with HPS exhibits sub-linear growth in memory requirements compared to parallel deployments of STL models, as the task-specific (dense) layers account for less than 2% of the total memory.

If we move to inference time (average time per sample), we can see that the MTL model is approximately 13% slower than a single STL one (0.052ms vs. 0.045ms), as given in Table 11. However, it provides a 3.4*x* improvement in the total inference time compared to the sequential execution of the four STL models (0.052ms vs. 0.18ms). These results indicate that the MTL model is more compact in terms of memory size and more efficient in inference time.

Progressing to Phase 2 requires validation to ensure that the KPImtl remains within the specified Δ tolerance of KPIstl. While the learning accuracy, memory requirements, and inference time of the MTL model are within the acceptable trade-off range compared to an STL model, it is observed that the inference time of the MTL model has decreased by 13%. However, it is important to consider the inference time in the context of sequential execution of all the STL models, as parallel execution would contradict the hardware constraint assumptions for the target device in Phase 2. As a result, the inference time of the MTL model still falls within this tolerance range even in scenarios where a low tolerance (e.g., $\Delta > 95\%$) is applied.

The inference time for both STL and MTL models will complement the KPImtl and KPIstl metrics, respectively, and it will be instrumental in Phase 2 for benchmarking the performance of the optimized models on the target device.

B. PHASE 2 STL AND MTL MODEL'S PERFORMANCE

As presented in Section V-A, we use TensorRT to optimize the resulting models from Phase 1 and deploy them on the NVIDIA Jetson TX2. Table 12 shows the performance evaluations of the optimized STL and MTL models in terms of total GPU memory requirements (buffered memory) for a batch of 64 L1 packets (3000 IQ samples), inference time, and accuracy per task when we use FP16 and FP32 precision. Notice that we did not explore lower precision reductions (e.g., INT8 via quantization) since Jetson TX2 does not support it.

For the optimized STL and MTL models using FP16, we reduced total memory requirements by up to 50% concerning the FP32 model's version (70MiB vs. 140MiB), as expected. Compared with the non-optimized versions (see Table 10), the improvement is even larger (up to 13%) thanks to the other optimization steps, e.g., layer fusion, that TensorRT

TABLE 12. Performance benchmark of optimized STL and MTL models using FP16 and FP32 (Phase 2). Inference time is per sample.

Model	Task	Precision	GPU Memory MiB	Inference Time x10^{-3}	Inference Time Aggregated x10^{-3}	Accuracy
STL	T0	FP16	70.33	0.105	0.43	0.995
	T1			0.115		1.00
	T2			0.104		0.955
	T3			0.106		0.896
	T0	FP32	140.66	0.150	0.61	0.995
	T1			0.149		1.00
	T2			0.153		0.955
	T3			0.158		0.896
MTL	T0	FP16	70.37	**0.119**		**0.999**
	T1					**1.00**
	T2					**0.973**
	T3					**0.903**
	T0	FP32	140.75	0.167		0.999
	T1					1.00
	T2					0.973
	T3					0.903

applied. *The resulting STL and MTL models reduce the memory requirements by a factor of* 2.65*x compared to their non-optimized version (186.72 vs. 70.37 and 189.2 vs. 70.37 MiB, respectively) and up to* 10.6*x compared to a parallel execution of them. Moreover, the accuracy remains consistent across both FP16 and FP32 models, indicating that the reduced precision has not compromised the accuracy of these tasks*.

Regarding inference time, we can see that the optimized MTL model running on the target device only increases its inference time by 13% with respect to an optimized STL using FP16 but outperforms it by a factor of 3.6*x* when the STL models are executed sequentially. Notice also that both STL and MTL models running on the target devices have a drop in performance compared to the non-optimized version running in server-grade hardware. This drop is expected based on the hardware capabilities of the server and the constrained device (see Table 6). However, if we focus on the MTL model using FP16, we can see that the *MTL model outperformed a sequential execution of the STL models running in a server by reducing its inference time by a factor of 3.42x (0.052ms vs. 0.180ms)*.

Notice that although the same MTL model is around 63% slower compared to a non-optimized version of a 1D-CNN STL model running on a server (0.045ms vs. 0.119ms), it is still an improvement compared to the inference time of a 2D-CNN STL model (0.119ms vs. 0.15ms, from Table 7). Moreover, according to [7, Secs. VI.A and VI.B], its inference time per packet is much lower than max_{exec}, which is (on average) 2ms for tasks 0 and 1, and 7ms for tasks 2 and 3.

In summary, Figure 9 shows a dual-axis comparison of average inference time in samples per ms and GPU memory consumption(in MiB between the FP16 and FP32 precision formats at various batch sizes. In general, a batch size of 64 provides the best trade-off between memory requirements, as it is aligned to the target device and inference time, which is below the maximum execution time for real-time processing. Moving from 64 to 128 batch size minimizes inference time from 0.119*ms* to 0.107*ms* but increases the memory in a 2*x* factor, from 70 to 140*MiB*.

C. PHASE 2 MTL MODEL'S ADAPTABILITY AND ENERGY EFFICIENCY ON DIFFERENT EDGE PLATFORMS

In this section, the energy consumption, calculated by using Equation (8), and energy efficiency, measured in joules per sample, of the resulting MTL models on the NVIDIA Jetson TX2 are presented. The performance results are compared with the well-known Raspberry Pi model 3B+ (RPI3B+) edge computing platform, both with and without the low-power Coral TPU USB Accelerator. Table 13 provides a summary of the hardware capabilities for both platforms. It is important to note that we had to recreate the MTL model using a 2D-CNN architecture using TensorFlow 2.15 and configure them to emulate 1D ones. This adjustment is necessary because the Coral TPU only supports models in TensorFlow Lite[15,16] format for inference and is incompatible with Conv1D layers. However, despite this modification, the resulting MTL model maintains an equivalent architecture and number of trainable parameters as the original 1D-CNN implemented in PyTorch, ensuring a fair comparison. Additionally, we employed quantization-aware training[17] with INT8 precision on the weights to compress the model, enabling it to run efficiently on constrained devices such as the RPI3B+. TensorFlow Lite served as the inference engine for the INT8 quantized models.

$$E = \sum_{i=1}^{n} P_i \cdot \Delta t_i \tag{8}$$

where:

- P_i is the power at time i,
- Δt_i is the time interval at time i, and
- n is the total number of time intervals.

Table 14 presents the performance evaluations regarding inference, energy consumption, and power consumption across various edge platforms, processing units, and inference engines. It is important to note that while the previous subsection focuses on the optimized MTL model using TensorRT as the inference engine on the NVIDIA Jetson TX2, the same model can also be deployed for inference using PyTorch's inference engine on both the Jetson and RPI3B+ CPUs, albeit with reduced performance.

[15] https://www.tensorflow.org/lite
[16] https://coral.ai/docs/edgetpu/tflite-python/
[17] https://www.tensorflow.org/model_optimization

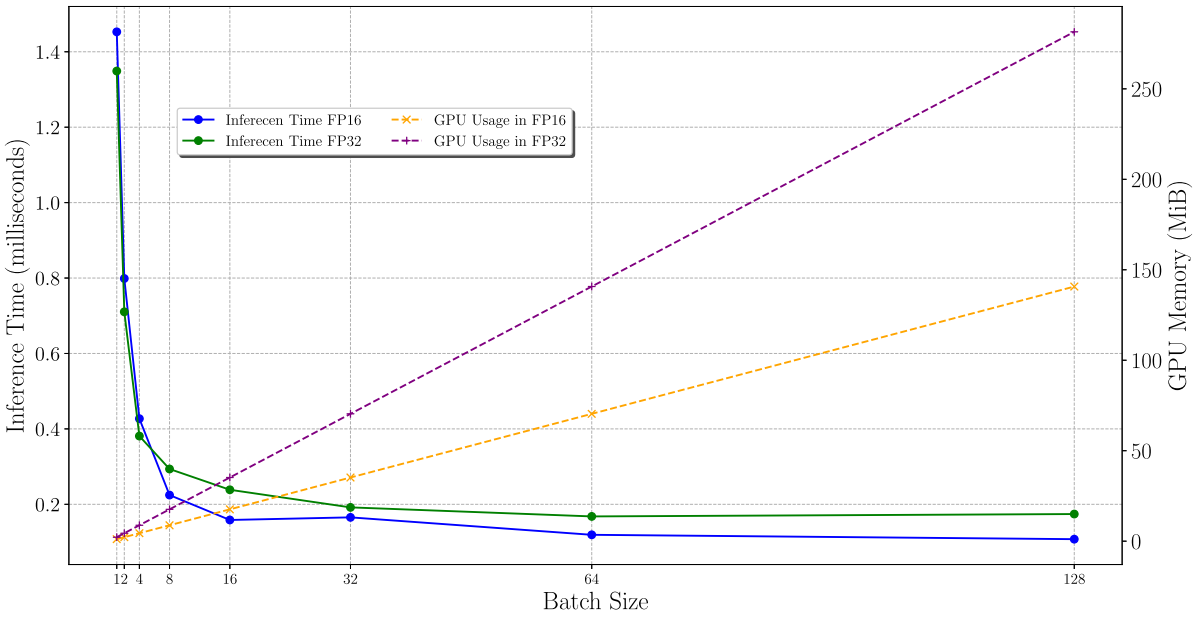

FIGURE 9. Comparison of inference time per sample and GPU memory usage as a function of batch size for FP16 and FP32 1D-CNN MTL models.

TABLE 13. Hardware specifications for Raspberry Pi 3 Model B+ and Coral TPU USB Accelerator.

Component	Specifications
Raspberry Pi 3 Model B+	
CPU	ARM Cortex-A53 (4 cores)
Memory type	1GB LPDDR2 SDRAM
GPU	Broadcom VideoCore IV
Max Power	2.5 Watts
Coral TPU USB Accelerator	
TOPS (INT8)	2 TOPS x Watt
Max Power	2 Watts (maximum)

Interestingly, the INT8 quantized model utilizing the TensorFlow Lite inference engine on the RPI3B+'s CPU outperforms, i.e., requires less energy, the model using the PyTorch inference engine with FP32 on the Jetson's CPU (54.8 vs. 82.7 millijoules/Sample), despite the Jetson CPU being superior to that of the RPI3B+. Furthermore, we can see that *the model optimized using TensorRT with FP16 and executed on the Jetson's GPU achieves a* 56*x improvement in energy efficiency that is* 0.97 *millijoules/Sample vs.* 54.8 *millijoules/Sample compared to any of the models running only on CPU*.

When utilizing the Coral TPU accelerator with the RPI3B+ to run the INT8 MTL models, we observe up to a 4.5*x* improvement in the energy efficiency compared to using only the CPU (11.9 millijoules/Sample vs. 54.8 millijoules/Sample). Interestingly, there were minor differences in the performance between the Coral TPU accelerator operating at standard (Coral TPU std) and maximum (Coral TPU max) current draw. One of the possible reasons for this discrepancy could be that the RPI3B+ might not provide sufficient current to the TPU when required, as minor variations were observed in both measurements.

Nevertheless, it is worth noting that *the model optimized using TensorRT with FP16 and executed on the Jetson's GPU achieves a* 12*x improvement in energy efficiency* (0.97 *millijoules/Sample vs.* 11.9 *millijoules/Sample) compared to any of the models running on the TPU*. This is despite the TPU being capable of performing more TOPS (up to 4) compared to the Jetson (1.33). The reason for the performance difference is that the TPU only supports batches of size 1 during inference due to its limited memory capacity. This results in additional overhead as data needs to be constantly transferred between the RPI3B+ memory and the TPU. In contrast, the Jetson has enough memory to support larger batches and utilizes shared memory between the CPU and GPU.

Complementing our previous point, Figure 10 illustrates the energy consumption of the MTL model on the Jetson TX2 across different batch sizes. It is evident that as the batch size increases, energy consumption decreases noticeably, indicating higher energy efficiency with larger batches. However, beyond a batch size of 16, the reduction in energy consumption becomes less pronounced, suggesting diminishing returns in efficiency improvements, consistent with the findings in Figure 9. This behavior can be attributed to energy consumption being proportional to inference time, which decreases sub-linearly with batches larger than 32 samples, as depicted in Figure 9.

Comparing the optimized MTL model with FP16 precision to the model with FP32, the former demonstrates an average reduction of 64% in energy consumption with batch sizes $\leq$ 16 (e.g., at batch size 64, energy consumption is 7.94 joules vs. 12.11 joules). Lastly, when comparing the energy consumption of the same model with the

TABLE 14. Performance comparison in terms of inference time, energy and power consumption, and energy efficiency of a Jetson TX2 and RPI3B+ with and without Coral TPU AI accelerator.

Platform	Inference engine	Processing Unit	Weights Precision	Total Inference time	Inference time per sample x10^{-3}	Energy Joules	Power Watts	Energy Efficiency Joules/Sample x10^{-3}
Jetson TX2	TensorRT	GPU	FP32	1.36	0.16	12.11	8.85	1.47
	TensorRT	**GPU**	**FP16**	**0.97**	**0.12**	**7.94**	**8.15**	**0.97**
	PyTorch	CPU	FP32	108.38	13.22	689.57	6.36	82.7
RPI3B+	PyTorch	CPU	FP32	288.45	35.21	924.94	3.21	112
	Tensorflow Lite	CPU	INT8	120.10	14.06	454.49	3.78	54.8
	Tensorflow Lite	Coral TPU std	INT8	30.56	3.73	99.36	3.25	11.9
	Tensorflow Lite	Coral TPU max	INT8	29.08	3.55	107.02	3.67	12.4

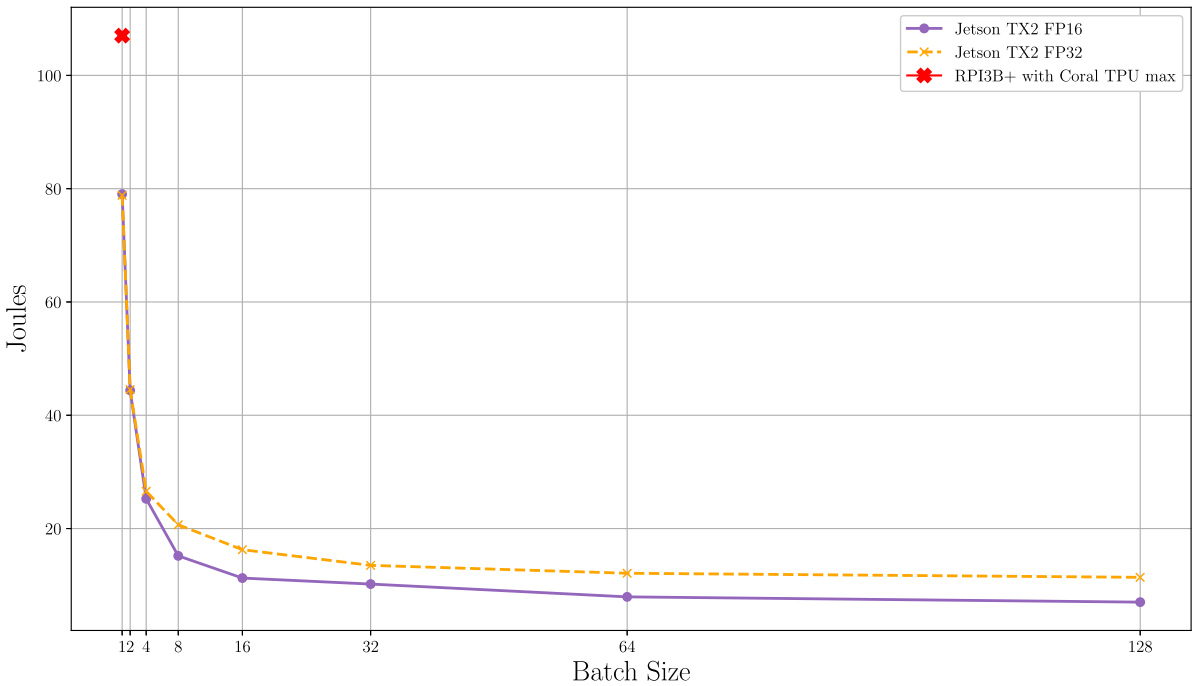

FIGURE 10. Comparison of energy consumption as a function of batch size for the 1D-CNN MTL models with FP16, FP32, and INT8 precision.

INT8 model running on the Coral TPU with batch size 1, the FP16 model on the Jetson exhibits approximately 20% lower energy consumption (79.04 joules vs. 99.36 joules). In general, we observe that the RPI3B+ with the TPU offers decent performance with lower power demands compared to the Jetson TX2 (3.67 Watts vs. 8.85 Watts) but exhibits lower energy efficiency (0.97 joules/sample vs. 11.9 joules/sample). This discrepancy can be attributed to limitations such as the TPU supporting only batch sizes of 1 and the RPI3B+ having limited CPU capacity compared to the Jetson TX2 (e.g., ARM Cortex-A53 vs. A57).

It is worth noting that the inference time of the RPI3B+ with the TPU (3.55ms) only meets the max_{exec} requirement for tasks 2 and 3, which is 7ms according to [7, Secs. VI.A and VI.B]. However, it will fail for tasks 0 and 1, which require 2ms. In this scenario, Phase 1 can be initiated to redesign the model or Phase 2 to apply other optimization techniques such as pruning, or even consider replacing the RPI3B+ with another edge platform, such as the newest Raspberry Pi 5 featuring the Broadcom BCM2712 quad-core Arm Cortex A76 processor at 2.4GHz.[18]

VII. CONCLUSION AND OPEN CHALLENGES

The paper introduces a novel methodology for designing spectrum-based TC systems optimized for constrained devices. This methodology integrates MTL with DNN optimization techniques, addressing the resource-intensive nature of previous state-of-the-art works. It provides both STL and MTL models tailored for limited-resource environments. Through extensive experimentation on an edge hardware platform such as the NVIDIA Jetson TX2, the study demonstrates that the designed MTL architecture, which combines 1D-CNN with DTP and HPS strategies, significantly enhances system efficiency.

[18]https://www.raspberrypi.com/products/raspberry-pi-5/

The application of DNN optimization methods, such as precision-reduction and layer-fusion, tailored to the device's capabilities, leads to a reduction in memory requirements by 2.65x times and improves execution time by 3.6x times compared to sequential execution of a non-optimized version of the STL models on a server-grade hardware platform. This was achieved while maintaining a minimal impact on accuracy (less than a 0.5% drop) with an energy efficiency of 0.97 millijoules per sample at inference. Compared to other edge platforms, such as the Raspberry Pi model 3B+ (RPI3B+) with the low-power AI accelerator Coral TPU, the NVIDIA Jetson achieves a 12-fold improvement in energy efficiency with no impact on accuracy.

While this work has successfully tackled several of the open challenges outlined in previous works, it is important to acknowledge that some persist, and new ones have emerged. As part of future research efforts, addressing these challenges is key to the development of resource-efficient, high-performance, and trustworthy spectrum-based TC systems.

Another important aspect to consider in future research is the increasing prevalence of encryption in modern networks. While the dataset used in this study included only L1 encryption, understanding how DL-based models can effectively classify encrypted traffic is essential for developing robust classification systems. This requires exploring innovative techniques and architectures capable of handling encrypted traffic patterns while maintaining high classification accuracy, similar to the research efforts in TC systems using byte-based packet representation [37], [55].

Additionally, enhancing the interpretability and explainability of these models is crucial for gaining insights into the decision-making process and fostering trust in their outcomes. This involves developing methodologies and tools to elucidate how the models arrive at their classifications, enabling users to understand and validate the reasoning behind the model's predictions. Potential frameworks for exploration could be based on those proposed for byte-based MTL TC [11] and AMC [56].

Addressing data privacy and security concerns while using spectrum-based TCs to handle user-generated network traffic presents significant challenges, primarily in balancing data utility with privacy in a real-world context where data volumes are vast. In byte-based TC systems, existing academic works try to achieve that balance via data perturbation, which is used to anonymize data while maintaining approximate distribution characteristics of the original dataset [57]. However, this can reduce data utility, which is crucial for classifier effectiveness. Restoring utility involves adjusting perturbed data to reflect the original attributes' order relationships, a process that requires complex manipulations to maintain privacy without sacrificing the accuracy of the TC.

Alternatively, federated learning offers a decentralized approach to training classifiers, enabling models to be trained directly on users' devices or local servers while preserving data privacy and increasing utility [58], [59]. Aggregating local updates from each device allows the model to learn from diverse data sources without directly accessing raw data while facilitating personalized recommendations or predictions for individual users. Therefore, exploring similar techniques in spectrum-based TCs is key to building trust with users and stakeholders and is crucial for ethical data handling.

Another important area for future research involves applying the methodology to a comprehensive end-to-end spectrum-based TC system for constrained devices. This process may encompass steps such as IQ sample capturing, packet assembly, and packet filtering, which potentially can be DNN-based (e.g., TR as described in [7]). Such an expansion would provide a more holistic understanding of the system's performance and applicability in real-world scenarios. Finally, the proposed methodology can be enhanced by integrating techniques such as neural architecture search [60] to automate the design of the DL architecture in phase one. However, evaluating the complexity of integrating such techniques and their impact on accelerating the design and development process has to be considered in future works.

REFERENCES

[1] K. Zia, A. Chiumento, and P. J. M. Havinga, "AI-enabled reliable QoS in multi-rat wireless IoT networks: Prospects, challenges, and future directions," *IEEE Open J. Commun. Soc.*, vol. 3, pp. 1906–1929, 2022.

[2] M. Z. Chowdhury, M. Shahjalal, S. Ahmed, and Y. M. Jang, "6G wireless communication systems: Applications, requirements, technologies, challenges, and research directions," *IEEE Open J. Commun. Soc.*, vol. 1, pp. 957–975, 2020.

[3] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1988–2014, 2nd Quart., 2019.

[4] A. Rago, G. Piro, G. Boggia, and P. Dini, "Multi-task learning at the mobile edge: An effective way to combine traffic classification and prediction," *IEEE Trans. Veh. Technol.*, vol. 69, no. 9, pp. 10362–10374, Sep. 2020.

[5] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges," *IEEE Trans. Netw. Service Manag.*, vol. 16, no. 2, pp. 445–458, Jun. 2019.

[6] S. Rezaei and X. Liu, "Deep learning for encrypted traffic classification: An overview," *IEEE Commun. Mag.*, vol. 57, no. 5, pp. 76–81, May 2019.

[7] M. Camelo, P. Soto, and S. Latré, "A general approach for traffic classification in wireless networks using deep learning," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 4, pp. 5044–5063, Dec. 2022.

[8] M. H. Rahman, R. B. Mofidul, and Y. M. Jang, "Spectrum based wireless radio traffic classification using hybrid deep neural network," in *Proc. 13th Int. Conf. Ubiquitous Future Netw. (ICUFN)*, 2022, pp. 95–99.

[9] M. Camelo, T. D. Schepper, P. Soto, J. Marquez-Barja, J. Famaey, and S. Latré, "Detection of traffic patterns in the radio spectrum for cognitive wireless network management," in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2020, pp. 1–6.

[10] Y. Zhang and Q. Yang, "A survey on multi-task learning," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 12, pp. 5586–5609, Dec. 2022.

[11] A. Nascita, A. Montieri, G. Aceto, D. Ciuonzo, V. Persico, and A. Pescapé, "Improving performance, reliability, and feasibility in multimodal multitask traffic classification with XAI," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 2, pp. 1267–1289, Jun. 2023.

[12] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "DISTILLER: Encrypted traffic classification via multimodal multitask deep learning," *J. Netw. Comput. Appl.*, vols. 183–184, Jun. 2021, Art. no. 102985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521000126

[13] M. Kanakis, "Designing efficient deep neural networks: Topological optimization, quantization and multi-task learning," Ph.D. dissertation, Departement Informationstechnologie und Elektrotechnik, Univ. Zürich, Zürich, Switzerland, 2023. [Online]. Available: https://ee.ethz.ch/de/

[14] S. Arish, S. Sinha, and K. G. Smitha, "Optimization of convolutional neural networks on resource constrained devices," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2019, pp. 19–24.

[15] D. Góez, P. Soto, S. Latré, N. Gaviria, and M. Camelo, "A methodology to design quantized deep neural networks for automatic modulation recognition," *Algorithms*, vol. 15, no. 12, p. 441, 2022.

[16] S. Kumar, R. Mahapatra, and A. Singh, "Automatic modulation recognition: An FPGA implementation," *IEEE Commun. Lett.*, vol. 26, no. 9, pp. 2062–2066, Sep. 2022.

[17] J. Fontaine, A. Shahid, B. Van Herbruggen, and E. De Poorter, "Impact of embedded deep learning optimizations for inference in wireless IoT use cases," *IEEE Internet Things Mag.*, vol. 5, no. 4, pp. 86–91, Dec. 2022.

[18] M. O. Demir, G. K. Kurt, and M. Karaca, "An energy consumption model for 802.11ac access points," in *Proc. 22nd Int. Conf. Softw., Telecommun. Comput. Netw. (SoftCOM)*, 2014, pp. 67–71.

[19] P. Silva, N. T. Almeida, and R. Campos, "A comprehensive study on enterprise Wi-Fi access points power consumption," *IEEE Access*, vol. 7, pp. 96841–96867, 2019.

[20] M. Miozzo, Z. Ali, L. Giupponi, and P. Dini, "Distributed and multi-task learning at the edge for energy efficient radio access networks," *IEEE Access*, vol. 9, pp. 12491–12505, 2021.

[21] A. Jagannath and J. Jagannath, "Multi-task learning approach for modulation and wireless signal classification for 5G and beyond: Edge deployment via model compression," *Phys. Commun.*, vol. 54, Oct. 2022, Art. no. 101793.

[22] P. Wang, X. Chen, F. Ye, and Z. Sun, "A survey of techniques for mobile service encrypted traffic classification using deep learning," *IEEE Access*, vol. 7, pp. 54024–54033, 2019.

[23] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," *Neurocomputing*, vol. 461, pp. 370–403, Oct. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221010894

[24] M. Hoyhtya, H. Sarvanko, M. Matinmikko, and A. Mammela, "Autocorrelation-based traffic pattern classification for cognitive radios," in *Proc. IEEE Veh. Technol. Conf. (VTC)*, 2011, pp. 1–5.

[25] E. Testi, E. Favarelli, and A. Giorgetti, "Machine learning for user traffic classification in wireless systems," in *Proc. 26th Eur. Signal Process. Conf. (EUSIPCO)*, 2018, pp. 2040–2044.

[26] C.-H. Liu, P. Pawelczak, and D. Cabric, "Primary user traffic classification in dynamic spectrum access networks," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 11, pp. 2237–2251, Nov. 2014.

[27] T. De Schepper, M. Camelo, J. Famaey, and S. Latré, "Traffic classification at the radio spectrum level using deep learning models trained with synthetic data," *Int. J. Netw. Manag.*, vol. 30, no. 4, 2020, Art. no. e2100.

[28] M. Girmay, V. Maglogiannis, D. Naudts, M. Aslam, A. Shahid, and I. Moerman, "Technology recognition and traffic characterization for wireless technologies in its band," *Veh. Commun.*, vol. 39, Feb. 2023, Art. no. 100563. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214209622001103

[29] T. J. O'Shea, S. Hitefield, and J. Corgan, "End-to-end radio traffic sequence recognition with recurrent neural networks," in *Proc. IEEE Glob. Conf. Signal Inf. Process. (GlobalSIP)*, 2016, pp. 277–281.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[31] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," in *Proc. 9th Int. Conf. Artif. Neural Netw. (ICANN)*, 1999, pp. 850–855.

[32] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 1310–1318.

[33] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," 2015, *arXiv:1506.00019*.

[34] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," 2018, *arXiv:1803.01271*.

[35] S. Rezaei and X. Liu, "Multitask learning for network traffic classification," in *Proc. 29th Int. Conf. Comput. Commun. Netw. (ICCCN)*, 2020, pp. 1–9.

[36] L. Liu, Y. Yu, Y. Wu, Z. Hui, J. Lin, and J. Hu, "Method for multi-task learning fusion network traffic classification to address small sample labels," *Sci. Rep.*, vol. 14, no. 1, p. 2518, 2024.

[37] W. Zheng, J. Zhong, Q. Zhang, and G. Zhao, "MTT: An efficient model for encrypted network traffic classification using multi-task transformer," *Appl. Intell.*, vol. 52, no. 9, pp. 10741–10756, Jul. 2022.

[38] L. Yang, S. Guo, D. Liu, Y. Zeng, X. Jiao, and Y. Zhou, "ConViTML: A convolutional vision transformer-based meta-learning framework for real-time edge network traffic classification," *IEEE Trans. Network Service Manag.*, early access, Mar. 29, 2024, doi: 10.1109/TNSM.2024.3383218.

[39] W. Wei, H. Gu, W. Deng, Z. Xiao, and X. Ren, "ABL-TC: A lightweight design for network traffic classification empowered by deep learning," *Neurocomputing*, vol. 489, pp. 333–344, Jun. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231222002673

[40] J. Cheng et al., "MATEC: A lightweight neural network for online encrypted traffic classification," *Comput. Netw.*, vol. 199, Nov. 2021, Art. no. 108472.

[41] M. Lu, B. Zhou, Z. Bu, K. Zhang, and Z. Ling, "Compressed network in network models for traffic classification," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, 2021, pp. 1–6.

[42] M. Lu, B. Zhou, and Z. Bu, "Two-stage distillation-aware compressed models for traffic classification," *IEEE Internet Things J.*, vol. 10, no. 16, pp. 14152–14166, Aug. 2023.

[43] R. Desislavov, F. Martínez-Plumed, and J. Hernández-Orallo, "Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning," *Sustain. Comput. Informat. Syst.*, vol. 38, Apr. 2023, Art. no. 100857. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537923000124

[44] S. Vandenhende, S. Georgoulis, W. Van Gansbeke, M. Proesmans, D. Dai, and L. Van Gool, "Multi-task learning for dense prediction tasks: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 7, pp. 3614–3633, Jul. 2022.

[45] S. Liu, E. Johns, and A. J. Davison, "End-to-end multi-task learning with attention," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2019, pp. 1871–1880. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2019.00197

[46] M. Guo, A. Haque, D.-A. Huang, S. Yeung, and L. Fei-Fei, "Dynamic task prioritization for multitask learning," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 270–287.

[47] O. Sener and V. Koltun, "Multi-task learning as multi-objective optimization," in *Proc. 32nd Adv. Neural Inf. Process. Syst.*, 2018, pp. 1–12.

[48] Z. Liu and D. Ding, "TensorRT acceleration based on deep learning OFDM channel compensation," *J. Phys. Conf. Series,*, vol. 2303, no. 1, 2022, Art. no. 012047.

[49] M. Blott et al., "Evaluation of optimized CNNs on heterogeneous accelerators using a novel benchmarking approach," *IEEE Trans. Comput.*, vol. 70, no. 10, pp. 1654–1669, Oct. 2021.

[50] Z. Liu, D. Ding, and Y. Fan, "Embedded hardware implementation of RFNoC-based OFDM communication system," in *Proc. 2nd Int. Conf. Artif. Intell., Autom., High-Perform. Comput. (AIAHPC)*, 2022, pp. 954–960. [Online]. Available: https://doi.org/10.1117/12.2641335

[51] *Precision—Quick Start Guide—Tensorrt Documentation*, Nvidia Softw. Co., Santa Clara, CA, USA, 2022, Accessed: Nov. 28, 2023.

[52] Y. Zhou and K. Yang. "Improved real-time deep learning inference by exploiting Tensorrt." 2023. [Online]. Available: http://dx.doi.org/10.2139/ssrn.4529548

[53] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha, "Deep learning inference parallelization on heterogeneous processors with TensorRT," *IEEE Embed. Syst. Lett.*, vol. 14, no. 1, pp. 15–18, Mar. 2022.

[54] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.

[55] J. Dai, X. Xu, H. Gao, X. Wang, and F. Xiao, "SHAPE: A simultaneous header and payload encoding model for encrypted traffic classification," *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 2, pp. 1993–2012, Jun. 2023.

[56] L. J. Wong and S. McPherson, "Explainable neural network-based modulation classification via concept bottleneck models," in *Proc. IEEE 11th Annu. Comput. Commun. Workshop Conf. (CCWC)*, 2021, pp. 0191–0196.
[57] Y. Lu, H. Tian, and J. Yu, "Privacy preservation for network traffic classification," in *Proc. 20th Int. Conf. Parallel Distrib. Comput., Appl. Technol. (PDCAT)*, 2019, pp. 84–89.
[58] C. L. Stergiou, K. E. Psannis, and B. B. Gupta, "InFeMo: Flexible big data management through a federated cloud system," *ACM Trans. Internet Technol.*, vol. 22, no. 2, pp. 1–22, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3426972
[59] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Int. Conf. Artif. Intell. Stat.*, 2017, pp. 1273–1282. [Online]. Available: https://proceedings.mlr.press/v54/mcmahan17a.html
[60] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 1997–2017, 2019.

DAVID GÓEZ received the bachelor's degree in telecommunications engineering and the M.Sc. degree in industrial automation and control from Metropolitan Technological Institute, Colombia, in 2011 and 2016, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science, University of Antwerp in association with IMEC. His main research focus is the implementation of deep learning models on resource-constrained devices intended for applications in 5G communications. Additionally, he has accumulated experience as a researcher in wireless communications, with special emphasis on software-defined radio.

ESRA AYCAN BEYAZIT received the M.Sc. degree in computer engineering from the İzmir Institute of Technology in 2008, and the Ph.D. degree in telecommunications from the İzmir Institute of Technology and the Conservatoire National des Arts et Métiers in 2016 through the Coutelle Ph.D. Program. She is a Senior Researcher of Telecommunication with the IDLab Research Group, University of Antwerp. She has both academical and industrial experience of more than six years. Her research interests are interference management, limited feedback links, heterogeneous networks, and artificial intelligence.

LUIS A. FLETSCHER received the B.S. degree in electronic and telecommunications engineering from the Universidad del Cauca, Popayán, Colombia, in 2001, the first M.Sc. degree in telematics from the Universidad de Murcia, Spain, in 2010, the second M.Sc. degree in telecommunications from Universidad Pontificia Bolivariana, Medellín, Colombia, in 2011, and the Ph.D. degree (cum laude) in engineering (energy systems) from the Universidad Nacional de Colombia (Medellín Campus) in 2018. Since 2012, he has been with the Electronic and Telecommunications Engineering Department, Universidad de Antioquia, Medellín, as an Associate Professor. His main research interests are energy efficiency of telecommunications systems, network management, and mobile networks planning.

JUAN F. BOTERO received the Ph.D. degree in telematics engineering from the Technical University of Catalonia, Spain, in 2013. In 2013, he joined the GITA Lab Research Group, Universidad de Antioquia, Colombia, where he is an Associate Professor with the Electronics and Telecommunications Engineering Department. His main research interests include quality of service, software-defined networks, NFV, cybersecurity, network management, and resource allocation.

NATALIA GAVIRIA received the electronic engineering degree from the University of Antioquia, Colombia, in 1996, the master's degree in electrical engineering from the Universidad de los Andes, Colombia, in 1999, and the Ph.D. degree in electrical and computer engineering from the University of Arizona, USA, in 2006. She is an Associate Professor with the University of Antioquia. She is an Active Member of the GITA Group. She has worked on theory and traffic modeling in wireless networks and on research projects in telemedicine.

STEVEN LATRÉ received the M.Sc. degree in computer science and the Ph.D. degree in computer science engineering from Ghent University, Belgium, in 2006 and 2011, respectively. He is currently the Vice President R&D of Artificial Intelligence with imec and a Professor with the University of Antwerp, Belgium. He has authored or coauthored over 200 papers published in international journals/conferences. His research expertise is on the intersection of machine learning and communication networks. He is a recipient of the IEEE COMSOC Award for the Best Ph.D. in Network and Service Management in 2012, the IEEE NOMS Young Professional Award in 2014, the IEEE COMSOC Young Professional Award in 2015, and the Laureate of the Belgian Academy in 2019.

MIGUEL CAMELO received the bachelor's degree in electronic engineering from the University of Ibague, Colombia, in 2006, the master's degree in systems and computer engineering from the University of Los Andes, Colombia, in 2010, and the Ph.D. degree in computer engineering from the University of Girona, Spain, in 2014. He is currently a Senior Researcher and leads the artificial intelligence (AI) for networks research track with the IDLab Research Group, University of Antwerp—imec, Belgium. He has authored or coauthored several papers in international conferences/journals. He has also participated and been awarded in international challenges in applied AI/ML in networks (e.g., DARPA SC2 and ITU AI/ML in 5G Challenge). His research interests are in the field of optimization and control of communication networks.