Big Data Goes Small: Real-Time Spectrum-Driven Embedded Wireless Networking Through Deep Learning in the RF Loop

Preprint · March 2019

CITATIONS
0

READS
31

2 authors:

Francesco Restuccia
Northeastern University
4 PUBLICATIONS **24** CITATIONS
SEE PROFILE

Tommaso Melodia
University at Buffalo, The State University of New York
185 PUBLICATIONS **9,591** CITATIONS
SEE PROFILE

Big Data Goes Small: Real-Time Spectrum-Driven Embedded Wireless Networking Through Deep Learning in the RF Loop

Francesco Restuccia and Tommaso Melodia
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115 USA
Email: {frestuc, melodia}@northeastern.edu

***Abstract*—The explosion of 5G networks and the Internet of Things will result in an exceptionally crowded RF environment, where techniques such as spectrum sharing and dynamic spectrum access will become essential components of the wireless communication process. In this vision, wireless devices must be able to (i) learn to autonomously extract knowledge from the spectrum on-the-fly; and (ii) react in real time to the inferred spectrum knowledge by appropriately changing communication parameters, including frequency band, symbol modulation, coding rate, among others. Traditional CPU-based machine learning suffers from high latency, and requires application-specific and computationally-intensive feature extraction/selection algorithms. Conversely, deep learning allows the analysis of massive amount of unprocessed spectrum data without ad-hoc feature extraction. So far, deep learning has been used for offline wireless spectrum analysis only. Therefore, additional research is needed to design systems that bring deep learning algorithms directly on the device's hardware and tightly intertwined with the RF components to enable real-time spectrum-driven decision-making at the physical layer. In this paper, we present *RFLearn*, the first system enabling spectrum knowledge extraction from unprocessed I/Q samples by deep learning directly in the RF loop. *RFLearn* provides (i) a complete hardware/software architecture where the CPU, radio transceiver and learning/actuation circuits are tightly connected for maximum performance; and (ii) a learning circuit design framework where the latency vs. hardware resource consumption trade-off can explored. We implement and evaluate the performance of *RFLearn* on custom software-defined radio built on a system-on-chip (SoC) ZYNQ-7000 device mounting AD9361 radio transceivers and VERT2450 antennas. We showcase the capabilities of *RFLearn* by applying it to solving the fundamental problems of modulation and OFDM parameter recognition. Experimental results reveal that *RFLearn* decreases latency and power by about 17x and 15x with respect to a software-based solution, with a comparatively low hardware resource consumption.**

I. Introduction and Motivation

Today's spectrum environment is exceptionally crowded. According to the latest Ericsson's mobility report, there are now 5.2 billion mobile broadband subscriptions worldwide, generating more that 130 exabytes per month of wireless traffic [1]. Moreover, it is expected that by 2020, over 50 billion devices will be absorbed into the Internet, generating a global network of "things" of dimensions never seen before [2].

Given that only few radio spectrum bands are available to wireless carriers [3], technologies such as radio-frequency (RF) spectrum sharing through beamforming [4–6], dynamic spectrum access [7–10] and anti-jamming technologies [11–13] will become essential in the near future. The first key challenge in enabling these systems is how to *effectively* and *efficiently* extract *meaningful* and *actionable* knowledge out of the tens of millions of in-phase/quadrature (I/Q) samples received every second by wireless devices. To give an example, to monitor a single 20 MHz WiFi channel, we need to process at least 40 million I/Q samples/s at Nyquist sampling rate. This generates a stream rate of about 2.56 Gbit/s, if samples are each stored in a 4-byte word. The second core challenge is that the RF channel is significantly time-varying (*i.e.*, in the order of milliseconds), which imposes strict timing constraints on the *validity* of the extracted RF knowledge. If (for example) the RF channel changes every 10ms, a knowledge extraction algorithm must run with latency (much) less than 10 ms to both (i) offer an accurate RF prediction and (ii) drive an appropriate physical-layer response to the inferred spectrum knowledge; for example, change in modulation/coding/beamforming vectors due to adverse channel conditions, local oscillator (LO) frequency due to spectrum reuse, and so forth.

To address the knowledge extraction challenge, *deep learning* [14] has been widely recognized as the technology of choice for solving classification problems for which no well-defined mathematical model exists. Deep learning goes beyond traditional low-dimensional machine learning techniques by enabling the analysis of unprocessed I/Q samples without the need of application-specific and computational-expensive feature extraction and selection algorithms [15]. Another core advantage is that deep learning architectures are application-insensitive, meaning that the same architecture can be retrained for different learning problems. For this reason, the Defense Advanced Research Projects Agency (DARPA), among other agencies in the United States, has recently launched the novel radio-frequency machine learning systems (RFMLS) research program [16], where the main objective is to fingerprint wireless devices by learning from RF data, rather than designing ad hoc systems hand-engineered by experts.

This paper has been accepted for publication in IEEE INFOCOM 2019.

The application of learning techniques to the RF domain presents several major hurdles that are substantially absent in traditional learning domains. Indeed, deep learning has been traditionally used in static contexts (*e.g.*, image and language classification [17, 18]), where the model latency is usually not a concern. *Another fundamental issue absent in traditional deep learning is the need to satisfy strict constraints on resource consumption.* Indeed, models with high number of neurons/layers/parameters will necessarily require additional hardware and energy consumption, which are clearly scarce resources in embedded systems.

Although prior work has investigated the opportunity of using learning [19–21] and deep learning [15, 22–24] techniques for RF spectrum analysis, we are not aware of practical demonstrations of real-time deep learning in the RF loop for spectrum-driven wireless networking on embedded systems. This is not without a reason. The core issue in enabling real-time deep spectrum learning on embedded devices is the existing lack of an embedded software/hardware architectural design where I/Q samples are directly read from the RF front-end and analyzed in real time on the device's hardware without CPU involvement. To further complicate matters, this architecture must also be flexible enough to be reconfigurable through software based on the wireless application's need. Finally, the strict constraints on latency and resource consumption (hardware and energy) imposed by the embedded RF domain necessarily require a design flow where learning performance is also met by energy/latency/hardware efficiency.

To fill this research gap, this paper makes the following core contributions:

• We propose *RFLearn*, the first learning-in-the-RF-loop system where spectrum-driven decisions are enabled through real-time deep learning algorithms implemented directly on the device hardware and operating on unprocessed I/Q samples. *RFLearn* provides (i) a full-fledged hardware architecture for system-on-chip (SoC) devices binding together CPU, radio transceiver and learning/actuation circuits for maximum performance (Section III); and (ii) a novel framework for RF deep learning circuit design that translates the learning model from a software-based implementation to an *RFLearn*-compliant circuit using high-level synthesis (HLS) (Section V), where the constraints on latency, energy, learning, and hardware performance can be tuned based on the application;

• We extensively evaluate *RFLearn* and its design cycle on a custom software radio composed of a Zynq-7000 SoC mounting AD9361 radio transceivers and VERT2450 antennas (Section VI). As a practical case study, we consider the fundamental problem of modulation and OFDM parameter recognition through deep learning [15], and train several classifier architectures to address it (Section VI-A). We experimentally compare the latency and power consumption performance of *RFLearn* with respect to the same model implemented in software (Section VI-B). We also apply our design framework to explore the tradeoff between HLS optimization and hardware consumption (Section VI-C). Experimental results indicate that *RFlearn* outperforms the software-based system by decreasing latency and power consumption by respectively 17x and 15x, with a relatively low hardware resource consumption.

II. Background Notions on Deep Learning

We use boldface upper and lower-case letters to denote matrices and column vectors, respectively. For a vector $\mathbf{x}$, x_i denotes the i-th element, $\|\mathbf{x}\|$ indicates the Euclidean norm, $\mathbf{x}^\intercal$ its transpose, and $\mathbf{x}\cdot\mathbf{y}$ the inner product of $\mathbf{x}$ and $\mathbf{y}$. For a matrix $\mathbf{H}$, H_{ij} will indicate the (i,j)-th element of $\mathbf{H}$. The notation $\mathcal{R}$ and $\mathcal{C}$ will indicate the set of real and complex numbers, respectively.

Deep neural networks are mostly implemented as multi-layer perceptrons (MLPs). More formally, an MLP with L layers is formally defined as a mapping $f(\mathbf{x}_i;\boldsymbol{\theta}) : \mathcal{R}^i \rightarrow \mathcal{R}^o$ of an input vector $\mathbf{x}_i \in \mathcal{R}^i$ to an output vector $\mathbf{x}_l \in \mathcal{R}^o$. The mapping happens through L subsequent transformations, as follows:

$$\mathbf{r}_j = f_j(\mathbf{r}_{j-1}, \theta_j) \qquad 0 \leq j \leq L \tag{1}$$

where $f_j(\mathbf{r}_{j-1}, \theta_j)$ is the mapping carried out by the j-th layer. The vector $\boldsymbol{\theta} = \{\theta_1, \ldots, \theta_L\}$ defines the whole set of parameters of the MLP.

A layer is said to be *fully-connected* (FCL) or *dense* if f_j has the form

$$f_j(\mathbf{r}_{j-1}, \theta_j) = \sigma(\mathbf{W}_j \cdot \mathbf{r}_{j-1} + \mathbf{b}_j) \tag{2}$$

where σ is an *activation* function, $\mathbf{W}_j$ is the *weight* matrix and $\mathbf{b}_j$ is the *bias* vector. This function introduces a non-linearity in the mapping processing, which allows for ever complex mappings as multiple layers are stacked on top of each other. Examples of activation functions are linear, *i.e.*, $\sigma(\mathbf{x})_i = x_i$, rectified linear unit (RLU), *i.e.*, $\sigma(\mathbf{x})_i = \max(0, x_i)$, and so on. Deep neural networks are generally trained using labeled training data, *i.e.*, a set of input-output vector pairs $(\mathbf{x}_{0,i}, \mathbf{x}^*_{L,i}), 1 \leq i \leq |\mathbf{S}|$, where $\mathbf{x}^*_{L,i}$ is the desired output of the neural network when $\mathbf{x}_{0,i}$ is used as input.

Convolutional layers (CVLs) [25] address the lack of scalability of FCLs by binding adjacent shifts of the same weights together similar to a filter sliding across an input vector. More formally, a CVL consists of a set of F filters $\mathbf{Q}_f \in \mathbb{R}^{h\times w}, 1 \leq f \leq F$, where F is also called the layer depth. Each filter generates a *feature map* $\mathbf{Y}^f \in \mathbb{R}^{n'\times m'}$ from an input matrix $\mathbf{X} \in \mathbb{R}^{n\times m}$ according to the following[1]:

$$Y^f_{i,j} = \sum_{k=0}^{h-1}\sum_{\ell=0}^{w-1} Q^f_{h-k,w-\ell} \cdot X_{1+s\cdot(i-1)-k,1+s\cdot(j-1)-\ell} \tag{3}$$

where $s \geq 1$ is an integer parameter called *stride*, $n' = 1 + \lfloor n + h - 2 \rfloor$ and $m' = 1 + \lfloor m + b - 2 \rfloor$. The matrix $\mathbf{X}$ is assumed to be padded with zeros, *i.e.*, $X_{ij} = 0\ \forall i \notin [1, n],\ j \notin [1, m]$. The output dimensions can be reduced by either increasing the stride s or by adding a *pooling layer* (POL). The POL computes a single value out of $p \times p$ regions of $\mathbf{Y}$, usually maximum or average value. For more details on CNNs, the reader may refer to [26].

[1]For simplicity, (3) assumes input and filter dimension equal to 2. This formula can be generalized for tensors having dimension greater than 2.

CNNs are commonly made up of only four layer types: convolutional (CVL), pooling (POL), fully-connected (FCL), and rectified-linear (RLL). The most common CNN architectures stack a number of CVL-RLU layers, (optionally) followed by POL layers, and repeat this pattern until the input has been merged spatially to a small size. At some point, it is common to transition to FCLs, with the last FCL holding the output (i.e., the classification output). In other words, the most common CNN architectures follow the pattern below:

$$\text{IN} \rightarrow \underbrace{[\overbrace{[\text{CVL} \rightarrow \text{RLL}]}^{N} \rightarrow \overbrace{\text{POL}}^{0\ldots P}]}_{1\ldots M} \rightarrow \overbrace{[\text{FCL} \rightarrow \text{RLL}]}^{1\ldots K} \rightarrow \text{FCL}$$

where N, M and K need to be chosen according to the specific classification problem. In computer vision applications, the most common parameters used are $0 < N \leq 3$, $M \geq 0$, $0 \leq K \leq 3$ [17, 27, 28]. However, networks with very high number of N and K have been proposed to achieve better classification accuracy [29].

III. RFLearn Architecture

Figure 1 depicts a high-level overview of the architecture of the *RFLearn* system. Together with the RF front-end (hardware) and the wireless network stack (software), *RFLearn* complements a full-fledged reprogrammable software-defined radio architecture where learning is entirely done in the RF loop without CPU involvement.

We briefly introduce the SoC architecture in Section III-A, then we describe each and every component of the *RFLearn* system. To ease readability, we have depicted with a blue and yellow color respectively the reception (RX) and transmission (TX) flow of data through the architecture; moreover, configuration data flow has been depicted in black.

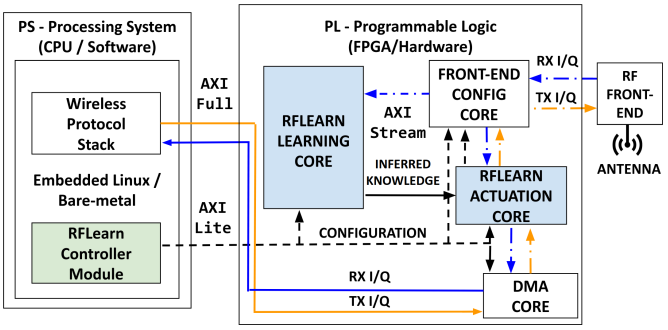

Fig. 1: The *RFLearn* Hardware Architecture.

A. RFLearn System-on-Chip Computer Architecture

RFLearn's architectural components entirely reside in the processing system (PS) and the programmable logic (PL) portions of a system-on-chip (SoC), which is an integrated circuit (also known as "IC" or "chip") that integrates all the components of a computer, *i.e.*, central processing unit (CPU), random access memory (RAM), input/output (I/O) ports and secondary storage (*e.g.*, SD card) – all on a single substrate [30]. We refer to SoCs thanks to their low power consumption [31] and because they allow the design and implementation of *customized hardware* on the field-programmable gate array (FPGA) portion of the chip, also called *programmable logic* (PL). Furthermore, SoCs bring unparalleled flexibility to *RFLearn*, as the PL can be reprogrammed at-will according to the desired learning design. The PL portion of the SoC can be managed by the *processing system* (PS), *i.e.*, the CPU, RAM, and associated buses.

RFLearn uses the *Advanced eXtensible Interface* (AXI) bus specification [32] to exchange data (i) between functional blocks inside the PL; and (ii) between the PS and PL. We use three AXI sub-specifications in *RFLearn*: *AXI-Lite*, *AXI-Stream* and *AXI-Full*. AXI-Lite is a lightweight, low-speed AXI protocol for register access, and it is used to configure the circuits inside the PL. AXI-Stream is used to transport data between circuits inside the PL. We use AXI-Stream since it provides (i) standard inter-block interfaces; and (ii) rate-insensitive design, since all the AXI-Stream interfaces share the same bus clock, the HLS design tool will handle the handshake between deep learning layers and insert FIFOs for buffering incoming/outgoing samples. AXI-Full is used to enable burst-based data transfer from PL to PS (and *vice versa*). Along with AXI-Full, *RFLearn* uses direct memory access (DMA) to allow PL circuits to read/write data obtained through AXI-Stream to the RAM residing in the PS. The use of DMA is crucial since the CPU would be fully occupied for the entire duration of the read/write operation, and thus unavailable to perform other work. Figure 1 depicts with continuous, dashed, and dot-dashed the AXI-Full, AXI-Lite and AXI-Stream interconnections.

With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller when the operation is done. This feature is useful when the CPU cannot keep up with the rate of data transfer (which happens very often in the case of RF samples processing).

B. PS Modules

In the following, we will refer to as *cores* (or *circuits*) and *modules*, respectively, the *RFLearn* components residing in the PL and PS.

The main challenge addressed by the PS is to provide modules that will drive and reconfigure the PL cores implementing the learning functionalities provided by *RFLearn*. The PS can run either on top an operating system (such as any embedded Linux distribution), or in "bare-metal" (also called "standalone") mode. In the latter, the only user application running on the CPU is the one specified at compile time. This mode is particularly useful to test the difference in latency between a learning system implemented in the PS (*i.e.*, software) and in the PL (*i.e.*, hardware).

Through the *RFLearn Controller* module, the PS has full domain over the activities of the cores residing in the PL. Specifically, the *Controller* is tasked to initialize/reconfigure through AXI-Lite (i) the RF front-end core with parameters such as sampling speed, center frequency, finite impulse response (FIR) filter taps, transmission (TX) and reception (RX) local oscillator (LO) frequency, TX/RX RF bandwidth, etc; and (ii) the *RFLearn* learning and actuation cores. The configuration values are stored in registers, so that both the PS and PL

cores can access the configuration through memory operations. Moreover, the *Controller* can, at any time, start/stop/check a PL core's operation through registers.

IV. The RFLearn PL Cores

The main objective of the PL cores is to provide a learning-in-the-RF-loop system where each and every physical-layer operation, included the real-time learning, is done in hardware, with minimum (or absent) involvement of the PS.

The physical-layer data exchange (*i.e.*, I/Q samples) between the PS and PL is handled as follows. The samples flow to/from the PL from/to the PS through the DMA core, which reads/-stores the samples from/into the RAM. The wireless protocol stack is tasked with programming the DMA according to its processing rate. However, the DMA can also be configured by the *Controller*, if no wireless protocol stack is present (*i.e.*, the system only processes physical-layer data).

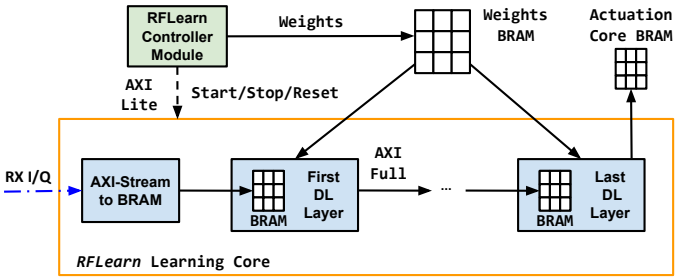

Fig. 2: The *RFLearn* Learning Core Architecture.

The PL receives/transmits I/Q samples through the *RF Front-end* core, which main operations can be summarized as follows: (i) down/up converts I/Q samples from the carrier frequency (for example, 2.4 GHz) to baseband; (ii) applies FIR filtering, DC and I/Q imbalance corrections; and (iii) sends the processed I/Q samples to the *RFLearn Learning* core through AXI-Stream. The I/Q samples received by the *RFLearn Actuation* core go through similar processing before being transmitted over the antenna. As mentioned before, the *RF Front-end* core parameters are set through AXI-Lite, and can be changed both PS-side (*i.e.*, by the *RFLearn* controller) and PL-side by the Actuation core.

The circuit that provides the deep learning capability to the system is the *RFLearn Learning* core, which architecture is detailed in Figure 2. The inputs to this core are (i) a number of unprocessed I/Q samples collected from the radio interface; and (ii) the parameters (*i.e.*, weights, filters, and so on) belonging to each layer (see Equations 1 and 3). Since the core may need to access these quantities in different time instants, both the I/Q samples and the weights are stored in block RAMs (BRAMs), a volatile memory that is implemented entirely in the PL portion of the SoC for maximum speed. Thus, the core necessitates a FIFO that converts the I/Q samples sent through AXI-Stream to a BRAM, so that the core can process the I/Q samples at its own pace. Transactions between the core and the BRAMs are done through AXI-Full.

Each layer presents the following structure: (i) it receives its input from a BRAM; (ii) it processes the input, according to the type of the layer (*i.e.*, convolutional, fully-connected, rectified linear unit, pooling); (iii) reads the weights from the weights BRAM; (iii) writes the result on the BRAM of the following layer. This architecture presents a number of major advantages: (a) modularity, since layers' computations are independent from each other; (b) scalability, since layers can be added on top of each other without changing the logic of the other layers; (c) reconfigurability, as weights can be changed by the *Controller* at any time *without need to change the hardware structure*. In Section V, we will discuss in details how this core is designed and optimized by using HLS.

The *RFLearn Actuation* core has the task to process the I/Q samples that are received/sent from/to the RF transceiver. Furthermore, the actuation core may (if needed) change the configuration of the RF transceiver itself (*e.g.*, change the FIR filters taps) and the modulation/demodulation logic (*i.e.*, change the physical-layer de-modulation process, increase the coding level, and so on). Since this core's functionality is highly dependent on the given application, we do not propose a specific architecture for it.

V. The RFLearn DL Core Design Framework

One the fundamental challenges addressed by *RFLearn* is how to transition from a software-based deep learning (DL) implementation (*e.g.*, developed with the Tensorflow [33] engine) to a hardware-based implementation compatible with the *RFLearn* architecture discussed in details in Section III. Basic notions of high-level synthesis and the *RFLearn* DL core design are presented in Sections V-A and V-B, respectively.

A. High-level Synthesis

RFLearn uses high-level synthesis (HLS) for its core designs. HLS is an automated design process that interprets an algorithmic description of a desired behavior (*e.g.*, C/C++) and creates a model written in hardware description language (HDL) that can be executed by the FPGA and implements the desired behavior [34].

Designing digital circuits using HLS has several advantages over traditional approaches. First, HLS programming models can implement almost any algorithm written in C/C++. This allows the developer to spend less time on the HDL code and focusing on the algorithmic portion of the design, and at the same time avoid bugs and increase efficiency, since HLS optimizes the circuit according to the system specifications.

The clock speed of today's FPGAs is several orders of magnitude slower than CPUs (*i.e.*, up to 200-300 MHz in the very best FPGAs). Thus, parallelizing the circuit's operations is crucial. In traditional HDL, transforming the signal processing algorithms to fit FPGA's parallel architecture requires challenging programming efforts. On the other hand, an HLS toolchain can tell how many cycles are needed for a circuit to generate all the outputs for a given input size, given a target parallelization level. This helps *RFLearn* to make the best trade-off between hardware complexity and latency.

Loop Pipelining. In high-level languages (such as C/C++) the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. *RFLearn* uses loop

pipelining to allows the operations in a loop to be implemented in a concurrent manner.

```
for (int i=0; i<2;i++) {
    Op_Read;      /* RD */
    Op_Execute;   /* EX */
    Op_Write;     /* WR */
}
```

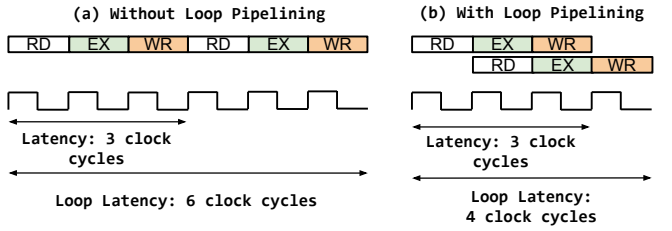

Fig. 3: Loop pipelining.

Figure 3 shows an example of loop pipelining, where a simple loop of three operations, *i.e.*, read (RD), execute (EX), and write (WR), is executed twice. For simplicity, we assume that each operation takes one clock cycle to complete. Without loop pipelining, the loop would take 6 clock cycles to complete. Conversely, with loop pipelining, the next RD operation is executed concurrently to the EX operation in the first loop iteration. This brings the total loop latency to 4 clock cycles. If the loop length were to increase to 100, then the latency decrease would be even more evident: 300 versus 103 clock cycles, corresponding to a speedup of about 65%.

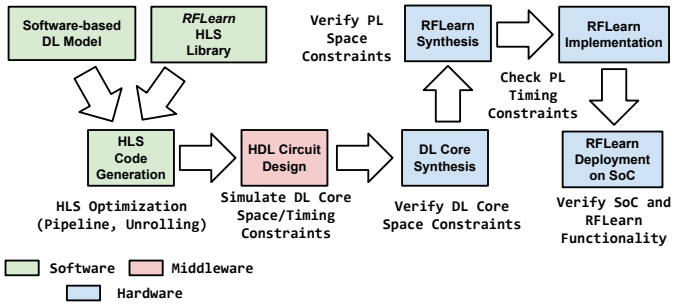

Fig. 4: The *RFLearn* DL Core Design Framework.

B. Design Steps

The framework presents several design and development steps, which are illustrated in Figure 4. Steps that involve hardware, middleware (*i.e.*, hardware description logic, or HDL), and software have been depicted with a blue, red, and green shade, respectively.

The first major step of the framework is to take an existing DL model and convert the model in HLS language, so it can be optimized and later on synthesized in hardware. Another critical challenge is how to make the hardware implementation fully reconfigurable, *i.e.*, the weights of the DL model may need to be changed by the *Controller* according to the specific training. To address these issues, *RFLearn* distinguishes between (i) the DL model architecture, which is the set of layers and hyper-parameters that compose the model itself, as in Equation (1); and (ii) the parameters of each layer, i.e., the neurons' and filters' weights (see Section II for details).

To generate the HLS code describing the software-based DL model, we leverage our own *RFLearn HLS Library*, which provides a set of HLS functions that parse the software-based DL model architecture and generates the HLS design corresponding to the architecture depicted in Figure 2. The *RFLearn HLS Library* currently supports the generation of convolutional (CVL), fully-connected (FCL), rectified linear unit (RLU), and pooling (POL) layers, and operated on fixed-point arithmetic for better latency and hardware resource consumption. The HLS code is subsequently translated to HDL code by an automated tool that takes into account optimization directives such as loop pipelining and loop unrolling. At this stage, the HDL describing the DL core can be simulated to (i) calculate the amount of PL resources consumed by the circuit (*i.e.*, flip-flops, BRAM blocks, etc); and (ii) estimate the circuit latency in terms of clock cycles.

After a compromise between space and latency as dictated by the application has been found, the DC core can be synthesized and integrated with the other PL components of *RFLearn*, and thus total space constraints can be verified. After implementation (*i.e.*, placing/routing), the PL timing constraints can be verified, and finally the whole *RFLearn* system can be deployed o and its functionality tested.

VI. Case Studies: Modulation and OFDM Parameters Recognition

To evaluate the performance of *RFLearn* on two real-world RF deep learning (DL) problems, we have considered the problem of physical-layer modulation recognition (in short, *ModRec*) and OFDM parameter recognition (in short, *OFDMRec*). We chose these problems since they are fundamental toward understanding ongoing wireless transmissions on a given portion of the spectrum.

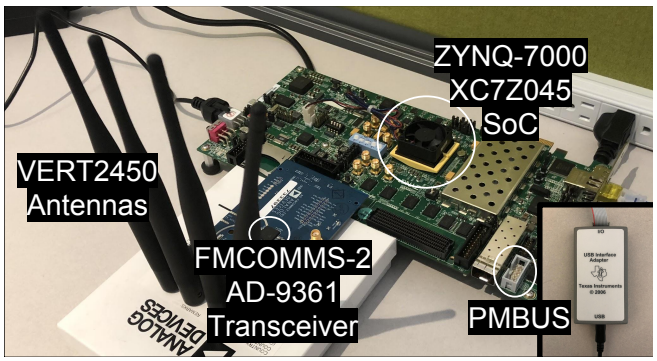

Fig. 5: The *RFLearn* Experimental Testbed.

For our experimental evaluation, we implemented the testbed shown in Figure 5 and composed of the following pieces of equipment: (i) a Xilinx Zynq-7000 XC7Z045-2FFG900C system-on-chip (SoC) with two ARM Cortex-A9 MPCore CPUs as processing system (PS) and a Kintex-7 FPGA as programmable logic (PL) [35], running on top of a Xilinx ZC706 evaluation board [36]; (ii) an Analog Devices (AD)-9361 RF transceiver [37] running on top of an AD-FMCOMMS2 evaluation board [38]; (iii) four VERT2450 antennas [39], two for

each TX/RX channel of the AD-9361; (iv) a Texas Instruments (TI) USB-TO-GPIO Interface Adapter to compute real-time power consumption of our board through the PMBUS standard [40]. We chose this equipment since it provides significant flexibility in the both the RF, PL and PS components, and thus allows us to fully evaluate the trade-offs during system design.

A. Deep Learning Model Training

As explained in Section V, the first step in the *RFLearn* system design process is to obtain a trained convolutional neural network (CNN) architecture. For this reason, we have performed a series of experiments with our testbed to obtain two datasets: (i) I/Q samples corresponding to 5 different modulation schemes (*i.e.*, BPSK, QPSK, 8PSK, 16QAM, DQPSK); and (ii) I/Q samples of an OFDM transmission with three FFT size parameters (*i.e.*, 64, 128, 256). To collect the samples, we have used another software-defined radio (*i.e.*, a Xilinx Zedboard [41] mounting an AD-FMCOMMS2 as RF transceiver) acting as transmitter, while our testbed was used to receive the samples.

If not stated otherwise, we train our model on inputs of size $32 \times 32 \times 2$ where $\ell = 32$, *i.e.*, 32 rows of 32 I samples plus 32 rows of 32 Q samples. We train the model using Tensorflow for 20 epochs, using 150,000 samples per class. We use as test set an additional dataset of 200,000 inputs generated from the collected experimental data. The filter and pooling length has been set to 3, and the filter stride to 1.

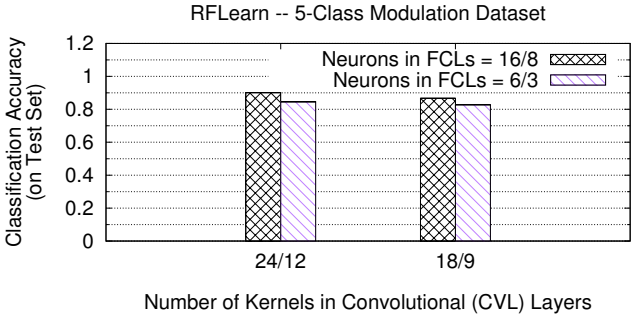

Fig. 6: 5-Class Modulation Dataset Accuracy Results.

To address *ModRec*, we consider an architecture with $\mathrm{M} = 2$ and $\mathrm{K} = 2$, fixing $\mathrm{N} = 1$. Figure 6 shows the related classification accuracy. *It can be observed that with a relatively small DL architecture with low number of kernels/neurons (as compared to modern computer vision models [17]) we can achieve an accuracy of at least 90% over 5 classes.* This is also thanks to the shift-invariance property of CNNs. We can also conclude that the number of kernels and the number of neurons definitely impact the model's accuracy; by doubling the number of kernels and increasing the number of neurons from 6-3 to 16-8, we can increase the accuracy by about 14%.

To further investigate the impact that the different kinds of modulations have on the model's accuracy, we trained the same DL architecture on two sets of 4 modulation classes, namely *S1* = {BPSK, QPSK, 16QAM, 8PSK} and *S2* = {BPSK, QPSK, 16QAM, DQPSK}. Since in *S2* we consider two very similar modulations (*i.e.*, QPSK and DQPSK), we should expect worse classification accuracy with respect to *S1* with the same DL architecture.

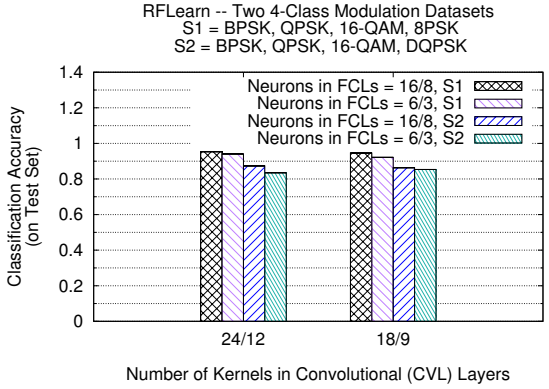

Fig. 7: Two 4-Class Modulation Datasets Accuracy Results.

Figure 7 shows the model's accuracy for both *S1* and *S2*. As expected, Figure 7 concludes that the model's accuracy is higher for *S1* than for *S2* (9% on average), since the classes are more distinct in the former case. Therefore, not only does the number of modulation classes impact the model's accuracy, but also the *type* of modulation classes considered.

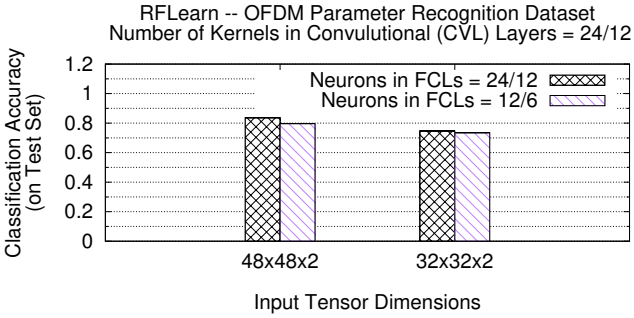

Fig. 8: OFDM Recognition Dataset Accuracy Results.

To investigate *OFDMRec*, we have trained an architecture with a greater number of kernels/neurons and also increased the input size. Figure 8 reports the classification results as a function of parameters and input size. *As we can see, by increasing the input size to 48x48 we can increase the accuracy by 10%, which concludes that an increase in model complexity increases classification accuracy accordingly.*

B. RFLearn vs. Software Latency/Space/Power Comparison

We now compare with our experimental testbed the *RFLearn* latency performance vs. a software (SW) implementation. To this end, we have used the *RFLearn HLS Library* to generate an equivalent model in C++ code to be executed in the PS portion of our testbed to test the difference in latency. To measure latency down to the clock cycle level, we have used an additional *AXI Timer* core [42] to count the number of clock cycles needed to produce the classification result in both hardware and software. To allow a fair comparison between the SW and the HW version, the testbed was run in "baremetal" mode (*i.e.*, without operating system).

In the following experiments, we set the PL clock frequency to 100 MHz (equivalent to 10ns clock period), with the exception of the RF front-end core that is clocked at 200 MHz. Note that the frequency of each PL clock can be changed at any time through register configuration, without the need to implement each core in the PL from scratch. The CPU clock speed is instead 667 MHz as per vendor datasheet.

Kern	Neur	SW	RFLearn	BRAM	LUT
24	16	235.8ms	13.7ms	166 (15%)	28247 (12%)
	8	220.1ms	13.2ms	166 (15%)	28227 (12%)
12	16	120.7ms	6.9ms	86 (7%)	20427 (9%)
	8	111.9ms	6.6ms	86 (7%)	20406 (9%)
6	16	61.1ms	3.4ms	46 (4%)	16413 (7%)
	8	56.5ms	3.2ms	46 (4%)	16399 (7%)

TABLE I: RFLearn/SW Comparison, $\mathrm{M} = 1$, $\mathrm{K} = 1$.

Tables I and II report the RFLearn vs. SW comparison in terms of latency (expressed in milliseconds), and the related HW resource consumption (with related percentage) in terms of number of BRAM and look-up tables (LUT), for the $\mathrm{M} = 1$, $\mathrm{K} = 1$ and $\mathrm{M} = 2$, $\mathrm{K} = 2$ architectures, respectively. For the sake of space, we do not report the number of flip-flops (FF) consumed since it is about 1% of the total resources in all the considered cases. The number of DSP48E1[2] slices [43] consumed was 21 and 39 out of 900, respectively. For each SW latency measurement, we report the average over 100 repetitions. We do not report standard deviations since they are below 1% of the average.

Kern	Neur	SW	RFLearn	BRAM	LUT
24-12	16-8	1376.4ms	75.9ms	220 (20%)	23673 (10%)
	6-3	1334.1ms	75.6ms	220 (20%)	23677 (10%)
18-9	16-8	767.8ms	45.2ms	220 (20%)	21738 (9%)
	6-3	795.2ms	44.9ms	220 (20%)	21689 (9%)
12-6	16-8	389.17ms	22.3ms	116 (10%)	19636 (8%)
	6-3	380.86ms	22.1ms	116 (10%)	19663 (8%)

TABLE II: RFLearn/SW Comparison, $\mathrm{M} = 2$, $\mathrm{K} = 2$.

The first important result to remark is the significant different in latency performance between *RFLearn* and SW. On the average, when $\mathrm{M} = 1$, $\mathrm{K} = 1$, *RFLearn* improves the latency by about 17x, *i.e.*, an order of magnitude with respect to SW, with a tolerable BRAM and LUT occupation of 15% and 12% in the worst case, respectively. The latency improvement brought by *RFLearn* is confirmed also in the $\mathrm{M} = 2$, $\mathrm{K} = 2$ experiments , where the latency improvement with respect to SW continues to be about 17x on the average, at the cost of an increase in HW resource consumption (20% vs 15% BRAM in the worst case). Surprisingly enough, in some cases *RFLearn* consumes less LUT resources when $\mathrm{M} = 2$, $\mathrm{K} = 2$. This can be explained by the fact that in these cases the *Flatten* layer (used to transform a tensor input to a linear input to the FCL) has less inputs than with $\mathrm{M} = 1$, $\mathrm{K} = 1$, which causes less LUT consumption.

[2]A DSP48E1 is a complex circuit providing a multiplier, an accumulator, a pre-adder, and two arithmetic logic unit, among other features.

Exp	1.0V	1.8V	1.5V	2.5V	3.3V	Total
Idle	0.16A	0.06A	0.02A	0.11A	0.06V	0.771W
Software	0.28A	0.12A	0.03A	0.11A	0.06V	1.014W
RFLearn	0.37A	0.13A	0.03A	0.13A	0.06A	1.172W

TABLE III: RFLearn/SW/Idle Power Comparison.

Table III summarizes the current absorption (in Amperes) as measured at the different power rails of the ZC706 board. To obtain these results, we selected the 24-12-16-8 *RFlearn* model (the most complex and thus, the worst case for power consumption) and averaged the results over 1000 measurements. As expected, *RFLearn* experiences higher power consumption than the software-based implementation. However, the lower latency (75.9 ms vs 1376.4 ms) experienced by *RFLearn* allows outstanding energy savings with respect to software. For example, in the considered case, the *RFLearn* energy consumption is 87.9 mJ, which is about 15x lower than software (1395.6 mJ).

C. HLS Latency Optimization

We have mentioned in Section V that *RFLearn* is capable to decrease drastically the latency of the DL learning core through HLS optimization, at the cost of an increase in HW consumption. To prove this point, Table IV shows the decrease in latency for different DL architectures upon HLS optimization, and the related amount of DSP48E1 slices consumed by the circuit. We do not report the increase in BRAM, LUT and FF since it was less than 1% in all cases.

Kern	Neur	Latency	DSP48E1
24	16	13.7ms $\rightarrow$ 8.2ms (-67%)	39 $\rightarrow$ 75 (+92%)
3	16	1.6ms $\rightarrow$ 1.04ms (-54%)	
24-12	16-8	75.9ms $\rightarrow$ 37.9ms (-100%)	21 $\rightarrow$ 39 (+85%)
12-6	16-8	22.3ms $\rightarrow$ 11.5ms (-93%)	

TABLE IV: *RFLearn* Optimization, Latency vs. HW Space.

The optimization made through HLS was to pipeline the loops corresponding to the computation of one filter output, so that the summing operations in Equation 3 can be executed in parallel. Table IV shows that by pipelining the convolution loops, we can achieve a significant reduction in latency. We point out that the decrease in latency becomes ever more evident as (i) the number of convolutional layers (CVLs) and (ii) the number of kernels in one layer increase. Indeed, we have a 67% vs. 100% latency reduction when going from one to two CVLs, and a 67% vs. 54% by going from 24 to 3 kernels. Obviously, this decrease in latency corresponds to an increase in DSP48E1 circuitry, which is almost double in the first architecture. Although the SoC considered in this paper supports up to 900 DSP48E1s, other architectures might have less DSP circuitry. Therefore, the trade-off between space and latency must always be considered before deploying the architecture on the SoC.

VII. Related Work

The usage of supervised machine learning techniques to interpret the wireless spectrum has been extensively investigated over the last few years; the reader can refer to [44–46]

for excellent surveys on the topic. Most of existing work is based on traditional low-dimensional machine learning [19–21, 47, 48], which requires (i) extraction and careful selection of complex features from the RF waveform (*i.e.*, average, median, kurtosis, skewness, high-order cyclic moments, etc.); and (ii) the establishment of tight decision bounds between classes based on the current application, which are derived either from mathematical analysis or by learning a carefully crafted dataset [49]. In other words, since feature-based machine learning is (a) significantly application-specific in nature; and (b) it introduces additional latency and computational burden due to feature extraction, its application to real-time hardware-based wireless spectrum analysis becomes unpractical, as the wireless radio hardware should be changed according to the specific application under consideration.

Recent advances in deep learning [14] have prompted researchers to investigate whether similar techniques can be used to analyze the sheer complexity of the wireless spectrum. For a compendium of existing research on the topic, the reader can refer to [50]. Among other advantages, deep learning is significantly amenable to be used for real-time hardware-based spectrum analysis, since different model architectures can be reused to different problems as long as weights and hyper-parameters can be changed through software. Among other issues, physical-layer modulation recognition through deep learning has received significant attention in the last two years [15, 22–24, 51, 52]. O'Shea *et al.* present in [15] several deep learning models to address the modulation recognition problem, while in [23] Karra *et al.* train hierarchical deep neural networks to identify data type, modulation class and modulation order. Kulin *et al.* present in [22] a conceptual framework for end-to-end wireless deep learning, followed by a comprehensive overview of the methodology for collecting spectrum data, designing wireless signal representations, forming training data and training deep neural networks for wireless signal classification tasks.

The core issue with prior approaches is that they leverage deep learning to perform offline spectrum analysis only. On the other hand, the opportunity of real-time hardware-based spectrum knowledge inference remains substantially uninvestigated. For this reason, this paper proposes a hardware architecture and learning core design strategy that together bring the power of deep learning *directly to the RF hardware loop*, which will enable sophisticated, real-time decision-making and knowledge inference with limited human intervention.

VIII. Conclusions

In this paper, we have proposed *RFLearn*, the first learning-in-the-RF-loop system that enables spectrum-driven decisions through real-time hardware-based I/Q deep learning algorithms. We have provided a complete hardware architecture for *RFLearn*, as well as a novel framework for RF deep learning circuit design that translates and optimizes the software-based implementation to produce a *RFLearn*-compliant circuit. We have extensively evaluated *RFLearn* on a practical testbed by considering the problem of modulation and OFDM parameter recognition through deep learning, and explored in details the trade-off between hardware space vs. latency and model complexity vs. accuracy. Furthermore, we have compared the latency performance of *RFLearn* with respect to a software-based system. Experimental results have demonstrated that *RFlearn* decreases the latency and power consumption by respectively 17x and 15x with a relatively low hardware resource consumption.

Acknowledgements

We sincerely thank the anonymous reviewers for their precious feedback, which helped us improve significantly the quality of our final manuscript. This work was supported in part by the National Science Foundation (NSF) under Grant CNS-1618727. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the U.S. Government.

References

[1] Ericsson Incorporated, "Ericsson Interim Mobility Report, February 2018," https://www.ericsson.com/assets/local/mobility-report/documents/2018/emr-interim-feb-2018.pdf, 2018.

[2] Cisco Systems, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016-2021 White Paper," http://tinyurl.com/zzo6766, 2017.

[3] Federal Communications Commission (FCC), "Spectrum Crunch," https://www.fcc.gov/general/spectrum-crunch.

[4] H. Shokri-Ghadikolaei, F. Boccardi, C. Fischione, G. Fodor, and M. Zorzi, "Spectrum sharing in mmwave cellular networks via cell association, coordination, and beamforming," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 11, pp. 2902–2917, 2016.

[5] M. A. Vázquez, L. Blanco, and A. I. Pérez-Neira, "Hybrid analog–digital transmit beamforming for spectrum sharing backhaul networks," *IEEE transactions on signal processing*, vol. 66, no. 9, p. 2273, 2018.

[6] L. Lv, J. Chen, Q. Ni, Z. Ding, and H. Jiang, "Cognitive non-orthogonal multiple access with cooperative relaying: A new wireless frontier for 5g spectrum sharing," *IEEE Communications Magazine*, vol. 56, no. 4, pp. 188–195, 2018.

[7] X. Jin, J. Sun, R. Zhang, Y. Zhang, and C. Zhang, "SpecGuard: spectrum misuse detection in dynamic spectrum access systems," *to appear, IEEE Transactions on Mobile Computing*, 2018.

[8] T. M. Chiwewe and G. P. Hancke, "Fast convergence cooperative dynamic spectrum access for cognitive radio networks," *IEEE Transactions on Industrial Informatics*, 2017.

[9] Federated Wireless, "Citizens Broadband Radio Service (CBRS) Shared Spectrum: An Overview," https://www.federatedwireless.com/wp-content/uploads/2017/09/CBRS-Spectrum-Sharing-Overview.pdf, 2018.

[10] S. Agarwal and S. De, "eDSA: energy-efficient dynamic spectrum access protocols for cognitive radio networks," *IEEE Transactions on Mobile Computing*, vol. 15, no. 12, pp. 3057–3071, 2016.

[11] L. Zhang, F. Restuccia, T. Melodia, and S. Pudlewski, "Learning to detect and mitigate cross-layer attacks in wireless networks: Framework and applications," in *Proc. of IEEE Conf. on Communications and Network Security*, Las Vegas, NV, USA, October 2017.

[12] J.-F. Huang, G.-Y. Chang, and J.-X. Huang, "Anti-jamming rendezvous scheme for cognitive radio networks," *IEEE Transactions on Mobile Computing*, vol. 16, no. 3, pp. 648–661, 2017.

[13] G.-Y. Chang, S.-Y. Wang, and Y.-X. Liu, "A jamming-resistant channel hopping scheme for cognitive radio networks," *IEEE Transactions on Wireless Communications*, vol. 16, no. 10, pp. 6712–6725, 2017.

[14] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, p. 436, 2015.

[15] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-air deep learning based radio signal classification," *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168–179, Feb 2018.

[16] Defense Advanced Research Projects Agency (DARPA), "The Radio Frequency Spectrum + Machine Learning = A New Wave in Radio Technology," https://www.darpa.mil/news-events/2017-08-11a, 2017.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[18] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[19] S. U. Pawar and J. F. Doherty, "Modulation recognition in continuous phase modulation using approximate entropy," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 843–852, Sept 2011.

[20] Q. Shi and Y. Karasawa, "Automatic modulation identification based on the probability density function of signal phase," *IEEE Transactions on Communications*, vol. 60, no. 4, pp. 1033–1044, April 2012.

[21] J. L. Xu, W. Su, and M. Zhou, "Software-defined radio equipped with rapid modulation recognition," *IEEE Transactions on Vehicular Technology*, vol. 59, no. 4, pp. 1659–1667, May 2010.

[22] M. Kulin, T. Kazaz, I. Moerman, and E. D. Poorter, "End-to-end learning from spectrum data: A deep learning approach for wireless signal identification in spectrum monitoring applications," *IEEE Access*, vol. 6, pp. 18 484–18 501, 2018.

[23] K. Karra, S. Kuzdeba, and J. Petersen, "Modulation recognition using hierarchical deep neural networks," in *Proc. of IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, Baltimore, MD, USA, March 2017, pp. 1–3.

[24] T. J. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, 2017.

[25] Y. LeCun *et al.*, "Generalization and network design strategies," *Connectionism in perspective*, pp. 143–155, 1989.

[26] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[27] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich *et al.*, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2015, pp. 1–9.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645.

[29] ——, "Deep residual learning for image recognition," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016, pp. 770–778.

[30] R. F. Molanes, J. J. Rodrguez-Andina, and J. Faria, "Performance characterization and design guidelines for efficient processor - FPGA communication in Cyclone V FPSoCs," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 5, pp. 4368–4377, May 2018.

[31] Pete Bennett (EE Times), "The Why, Where and What of Low-Power SoC Design," https://www.eetimes.com/document.asp?doc_id=1276973, 2004.

[32] Xilinx Inc., "AXI Reference Guide, UG761 (v13.1) March 7, 2011," https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, 2011.

[33] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[34] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using vivado hls," in *Proc. of International Conference on Field-Programmable Technology (FPT)*, Kyoto, Japan, 2013, pp. 362–365.

[35] Xilinx Inc., "Zynq-7000 SoC Data Sheet: Overview," https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, 2018.

[36] ——, "ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide," https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf, 2018.

[37] Analog Devices Incorporated, "AD9361 RF Agile Transceiver Data Sheet," http://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf, 2018.

[38] Analog Devices Inc., "AD-FMCOMMS2-EBZ User Guide," https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz, 2018.

[39] Ettus Research (A National Instrument Company), "VERT2450 Antenna," https://www.ettus.com/product/details/VERT2450, 2016.

[40] Texas Instruments Inc., "USB Interface Adapter Evaluation Module User's Guide," http://www.ti.com/lit/ml/sllu093/sllu093.pdf, 2006.

[41] Xilinx Inc., "Zedboard Hardware User Guide," http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf, 2018.

[42] ——, "AXI Timer v2.0 LogiCORE IP Product Guide," https://www.xilinx.com/support/documentation/ip_documentation/axi_timer/v2_0/pg079-axi-timer.pdf, 2018.

[43] ——, "7 Series DSP48E1 Slice User Guide," https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf, 2018.

[44] M. Bkassiny, Y. Li, and S. K. Jayaweera, "A survey on machine-learning techniques in cognitive radios," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1136–1159, Third 2013.

[45] C. Jiang, H. Zhang, Y. Ren, Z. Han, K. C. Chen, and L. Hanzo, "Machine learning paradigms for next-generation wireless networks," *IEEE Wireless Communications*, vol. 24, no. 2, pp. 98–105, April 2017.

[46] M. Chen, U. Challita, W. Saad, C. Yin, and M. Debbah, "Machine learning for wireless networks with artificial intelligence: A tutorial on neural networks," *CoRR*, vol. abs/1710.02913, 2017. [Online]. Available: http://arxiv.org/abs/1710.02913

[47] M. L. D. Wong and A. K. Nandi, "Automatic digital modulation recognition using spectral and statistical features with multi-layer perceptrons," in *Proceedings of the Sixth International Symposium on Signal Processing and its Applications (Cat.No.01EX467)*, vol. 2, 2001, pp. 390–393.

[48] S. Ghodeswar and P. G. Poonacha, "An SNR estimation based adaptive hierarchical modulation classification method to recognize M-ary QAM and M-ary PSK signals," in *Proc. of International Conference on Signal Processing, Communication and Networking (ICSCN)*, Chennai, India, March 2015, pp. 1–6.

[49] S. Shalev-Shwartz and S. Ben-David, *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[50] Q. Mao, F. Hu, and Q. Hao, "Deep learning for intelligent wireless networks: A comprehensive survey," *to appear, IEEE Communications Surveys & Tutorials*, 2018.

[51] T. Wang, C.-K. Wen, H. Wang, F. Gao, T. Jiang, and S. Jin, "Deep learning for wireless physical layer: Opportunities and challenges," *China Communications*, vol. 14, no. 11, pp. 92–111, 2017.

[52] N. E. West and T. O'Shea, "Deep architectures for modulation recognition," in *Proc. of IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, Baltimore, MD, USA, March 2017, pp. 1–6.